


1-1-2015

Building Computing-As-A-Service Mobile Cloud System

Kun Wang
Wayne State University,

Follow this and additional works at: https://digitalcommons.wayne.edu/oa_dissertations

 Part of the [Computer Engineering Commons](#), [Computer Sciences Commons](#), and the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Wang, Kun, "Building Computing-As-A-Service Mobile Cloud System" (2015). *Wayne State University Dissertations*. 1385.
https://digitalcommons.wayne.edu/oa_dissertations/1385

This Open Access Dissertation is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Dissertations by an authorized administrator of DigitalCommons@WayneState.

**BUILDING A COMPUTING-AS-A-SERVICE MOBILE CLOUD
SYSTEM**

by

KUN WANG

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2015

MAJOR: COMPUTER
 ENGINEERING

Approved By:

Advisor

Date

ACKNOWLEDGEMENTS

One of the most important decisions in my life was to pursue a PhD degree in the US. The Ph.D. study is full of adventures and challenges. I could not have achieved anything without the help from so many people in various ways. First and for foremost, I would like to thank my advisor, Dr. Cheng-Zhong Xu, for his invaluable guidance, and encouragement through-out my PhD journey. I am also grateful to my committee members: Dr. Song Jiang, Dr. Nabil Sarhan, and Dr. Lihao Xu for their time, interest, and insightful suggestions to improve this work.

I would like to specially thank my peer friend Jia Rao and his family for their selfless help through my PhD journey. I am also greatly thankful to my other colleagues and friends, Jiayu Gong, Xiangping Bu, Bo Yu, Zhen Kong, Yudi Wei, Xuechen Zhang, Yuehai Xu, Yizhe Wang, Ke Liu, Jianqiang Ou, Hao Zhou, Xingbo Wu and so many others. There are so many unforgettable beautiful memories with all of you. Thank you all and good luck with your future career.

Finally, this thesis would not be possible without the unconditional support from my parents, my brother, and my girlfriend. They are always my solid backing. I am deeply grateful for all their love, encouragement and understanding.

TABLE OF CONTENTS

Acknowledgements	ii
List of Figures	vii
List of Tables	ix
Chapter 1 INTRODUCTION	1
1.1 Background	2
1.1.1 Virtualization and Cloud Computing	2
1.1.2 Mobile Virtualization and Mobile Cloud Computing	4
1.2 Motivations	5
1.3 Substrate: Agile VM Deployment	8
1.4 SBCO: Efficient SMP VM Co-Scheduling	9
1.5 AMPhone: Flexible Mobile Augmentation	10
1.6 Organization	11
Chapter 2 RELATED WORK	13
2.1 Agile VM Deployment	13
2.2 SMP VM Co-Scheduling	15
2.3 Flexible Mobile Augmentation	19
Chapter 3 AGILE VIRTUAL MACHINE DEPLOYMENT	23
3.1 Introduction	23
3.2 Background	26

3.2.1	Cost of VM Creation	26
3.2.2	VM State Transition	27
3.2.3	Challenges of Rapid VM Deployment	28
3.3	Design of VM Substrate	29
3.3.1	VM Substrate and Pool	30
3.3.2	VM Clone From Substrate	33
3.3.3	VM Substrate Generation	34
3.3.4	VM Fork	35
3.4	Implementation	38
3.4.1	Resource Shrinking and Expanding	38
3.4.2	Substrate Multicast and Compression	49
3.5	Evaluation	52
3.5.1	Overhead	52
3.5.2	Performance Comparison	54
3.6	Conclusions and Future Work	56
Chapter 4 EFFICIENT SMP VIRTUAL MACHINE SCHEDULING		59
4.1	Introduction	59
4.2	Background	62
4.3	Challenges	64
4.3.1	Dynamic vCPU Affinity	64
4.3.2	Costly vCPU Context Switch	67
4.3.3	The Effect of Scheduling Distance	68
4.4	Self-Boosted Co-Scheduling	71
4.4.1	Overview	72

4.4.2	Extending Red Black Tree	73
4.4.3	SBCO Algorithm	76
4.4.4	Performance Considerations	77
4.5	Implementation	80
4.6	Evaluation	82
4.6.1	Experiment Design	82
4.6.2	Experimental Results	84
4.7	Summary	91
Chapter 5 FLEXIBLE MOBILE AUGMENTATION		93
5.1	Introduction	93
5.2	Background and Challenges	95
5.2.1	Mobile Computing and Cloud	95
5.2.2	Architecture Compatibility	97
5.2.3	Cost of Communication and Computing	98
5.3	System Design	100
5.3.1	Overview	100
5.3.2	Whole System Replication	101
5.3.3	Mobile Augmentation	102
5.3.4	Incremental Synchronization	102
5.3.5	Limitations	104
5.4	Implementation	105
5.5	Evaluation	106
5.5.1	Experimental Setup	106
5.5.2	Evaluation Results	108

5.6 Discussion and Summary	111
Chapter 6 CONCLUSIONS	114
6.1 Conclusions	114
6.2 Future Directions	115
References	117
Abstract	130
Autobiographical Statement	132

LIST OF FIGURES

Figure 3.1	VM State Transition.	28
Figure 3.2	Create VM from substrate.	32
Figure 3.3	VM dock to substrate.	36
Figure 3.4	VM's memory footprint.	39
Figure 3.5	Vertical COW.	44
Figure 3.6	Linear COW.	44
Figure 3.7	IO sync.	46
Figure 3.8	Impact of shrinking degree.	48
Figure 3.9	Compression cost.	49
Figure 3.10	Effectiveness of multicast.	50
Figure 3.11	Time breakdown of VM creation.	54
Figure 3.12	Delay time of offloading checker.	55
Figure 3.13	Startup time comparison.	55
Figure 4.1	Cost of vCPU context switch w/ and w/o virtualization.	68
Figure 4.2	Cost of vCPU context switch w/ and w/o virtualization.	69
Figure 4.3	Scheduling distance sibling vCPUs.	72
Figure 4.4	CFS run queue snapshot.	73
Figure 4.5	Extended red black tree with index.	73
Figure 4.6	Average runtime of kernbench.	84

Figure 4.7	Context switches numbers of kernbench in Log scale.	85
Figure 4.8	Parsec performance with average rq size is eight.	85
Figure 4.9	Parsec performance with average rq size is twelve.	86
Figure 4.10	Performance of SPECjbb benchmark.	89
Figure 4.11	Scalability of SBCO on Parsec workloads.	90
Figure 4.12	Relative Standard Deviation(RSD) of the Maximum Absolute Lag(MAL) of each VM.	91
Figure 5.1	System Architecture.	101
Figure 5.2	Virtual Phone.	103
Figure 5.3	lmbench performance.	108
Figure 5.4	Octane performance.	109
Figure 5.5	Openssl performance.	111

LIST OF TABLES

Table 3.1	Cost of creating VM from templates.	25
Table 4.1	Statistics of spin-lock Usage	63
Table 4.2	Probability of stacking sibling vCPUs	67
Table 4.3	Probability of scheduling distance exceeds the threshold	72

Chapter 1:INTRODUCTION

The last five years have witnessed the proliferation of smart mobile devices, the explosion of various mobile applications and the rapid adoption of cloud computing in business, governmental and educational IT deployment. There is also a growing trends of combining mobile computing and cloud computing as a new popular computing paradigm nowadays. This thesis envisions the future of mobile computing which is primarily affected by following three trends: First, servers in cloud equipped with high speed multi-core technology have been the main stream today. Meanwhile, ARM processor powered servers is growingly became popular recently and the virtualization on ARM systems is also gaining wide ranges of attentions nowadays. Second, high-speed internet has been pervasive and highly available. Mobile devices are able to connect to cloud anytime and anywhere. Third, cloud computing is reshaping the way of using computing resources. The classic pay/scale-as-you-go model allows hardware resources to be optimally allocated and well-managed.

These three trends lend credence to a new mobile computing model with the combination of resource-rich cloud and less powerful mobile devices. In this model, mobile devices run the core virtualization hypervisor with virtualized phone instances, allowing for pervasive access to more powerful, highly-available phone clones in the cloud. The centralized cloud, powered by rich computing and memory recourses, hosts virtual phone clones and repeatedly synchronize the data changes with virtual phone instances running on mobile devices. Mobile users can flexibly isolate different computing environments. For example, mobile users are able to switch between a personal phone and a corporate phone by trigger the migration of a virtual phone

instance to their devices or share their phone or media data by directly applying the changes in the cloud. While sounding admittedly utopian, this mobile cloud computing model has the capability to redefine the access scheme between mobile devices and cloud. As a result, this vision endows users with flexible architecture for the seamless access to ambient cloud resources to boost mobile applications, making them fast and energy efficient.

In this chapter, we introduce the background of virtualization and cloud computing, as well as the the ongoing shifting virtualization to mobile devices. Then we discuss the motivation of this dissertation work, present an overview of our solution and outline the organization of this dissertation.

1.1 Background

1.1.1 Virtualization and Cloud Computing

Virtualization is one of the key enabling technology of cloud computing. It is a combination of software and hardware aggregation and partitioning that creates sandbox virtual machines(VMs) and multiplex hardware resources to present one or many computing environments. The sandbox abstraction gives guest operating system(OS) the illusion that is is running on top dedicated hardware. Virtualization endows users with security, isolation features and full control of their own VM sandbox. It has been widely used for server consolidation, supporting multiple operating systems, securing cloud computing, system level development and debugging.

With virtualization, service providers offers great flexibility of using the data center resources. There are three representative type of cloud services: Platform as a Service (PaaS), Software as a Service (SaaS) and Infrastructure as a Service (IaaS).

PaaS offers the options for users to select preferable operating systems to deploy personal softwares in the cloud. SaaS allows users to access the software applications services in the cloud without concerning hardware maintenance and system configuration. In IaaS, raw data center hardware, such as CPU, memory, storage and network, are offered to cloud users as configurable virtual machines on the fly. Thus cloud users can start with small scale of VMs and expand to large scale on demand, giving users the illusion of infinite, elastic and pay-per-use computing resources are available. In this model, cloud users have more control on the available computing resources and avoid the initial over-provisioning hardware resource, which is often needed during the peak demand.

Besides the scale flexibility offered by cloud services, cloud providers can greatly increase the data center resource utilization efficiency by consolidating multiple VMs onto one physical machine, assuming that the interference between VMs has only limited impact on applications in the base load. Many applications nowadays are intrinsically designed with the capability for easy scale. Consequently, it is very common that one VM is running one specific application such as database or web server. In order to avoid the overlap of same type of resource competition, cloud providers can optimize VM consolidation based on the type of resources are consumed. For example, a VM with memory intensive applications and a VM that runs CPU intensive applications can be consolidated to the same physical machine. These VMs can also be seamlessly migrated to other physical machines or even other data centers when system upgrade or scale expansion is needed or in the cases when faults or failures happen. To achieve such scale and location flexibility and improve resource utilization, cloud providers need effective virtualized resource management mechanisms.

1.1.2 Mobile Virtualization and Mobile Cloud Computing

As the proliferation of mobile devices and the advances in mobile processor performance, memory and storage capacities, there is a growing trend of applying many of the previous desktop and server virtualization techniques to mobile virtualization on ARM-based devices due to ARM CPUs is the dominating option in mobile devices. Similar to the past revolutionary change in business, governmental and education IT deployment due to X86 virtualization, virtualization on mobile devices is also changing the way mobile users exchange and consume data. Today's mobile devices tends to have PC like features and functions, thus many mobile users favor the bring your own device(BYOD) model and tends carry multiple phones to accommodate work to access privileged enterprise content maintained in the cloud or daily personal needs, by convenience or by security restrictions imposed by corporate IT departments. Mobile virtualization enables multiple mobile operating systems to run on the same device, simultaneously addressing the security restrictions that required by the corporate entities, as well as isolating the IP of open source OSes from proprietary offerings. The virtualization of mobile devices offers significant potential in addressing the mobile manageability, security, cost, compliance, application development and deployment challenges that exist in the enterprise today.

With the advancement of 4G and LTE wireless network infrastructure, mobile devices are able to access to cloud at any place and any time. Such convergence of mobile computing and cloud computing emerges a new paradigm of mobile cloud computing. The unlimited resources in cloud free mobile devices from the constrain of processing power, memory and battery life. Mobile cloud computing was widely referred as the combination of mobile and cloud in the early days [97, 37], in which

applications are running in a remote data center with abundant resources while mobile devices act like thin clients connecting to remote cloud via wireless network. For example, there are many cloud service providers that offer online storage services to mobile users to overcome the limitations of storage space on mobile devices. Another popular definition of mobile cloud computing is to consider all mobile devices as components of cloud offering resources to build a mesh network [98]. Therefore, the connected mobile devices work as a cooperative computing unit to provide services like collective sensing. The cloudlet concept [79] is another representative approach of mobile cloud computing. In this model, mobile devices offload their workloads to a local cloudlet comprised of several multi-core computers with connectivity to the remote cloud servers.

Cloud computing is able to augment mobile devices' computing capability via remote execution or application offloading. For instance, the resource intensive components of mobile applications can be offloaded, in whole or part, to the resource-rich cloud [70, 24, 73, 18]. The efficiency of offloading approaches highly depends on the partitioning of the components and the communication with the remote server. Due to the sharp contrast of the processing power between cloud and mobile devices, such execution offloading can significantly improve the performance for certain type of applications. In general, cloud computing extends traditional mobile cloud applications with unlimited storage and computation resources as well as task-oriented services.

1.2 Motivations

Despite of continuous and still ongoing improvement in hardware computing capabilities of mobile devices, there are still some computing requirements of mobile users,

especially enterprise users, are not achieved. Many intrinsic restrictions of mobile devices encumber intense mobile computing. Mobile devices processing limitation due to slow processing speed and limited memory space remains one of the major challenges in mobile computing [78]. In contrast, cloud has abundant computing power in terms of CPU processing speed, memory size, and disk space. Therefore, leveraging cloud resource to augment mobile computing is a nature evolution. Additionally, most of mobile devices today are powered by replenishable lithium-ion battery that may last only few hours if device is involved in intensive computation. Executing mobile applications in cloud can significantly save the energy consumption on mobile devices.

Leveraging cloud resources to augment mobile computing is tradeoff between sacrificing the communication cost involved in remote execution and the augmentation gain when using cloud resources. The underlying assumption is that as long as execution in cloud is significantly more faster or more reliable than that on mobile devices, the augmentation gain is more than the cost paid and remote execution is still worth it. In practice, the cloud execution performance gain may vary with applications. In general, our design pushes for specialized hardware that has the compatible architecture with mobile devices to run virtualized mobile clones and customized software to support the communication between cloud and mobile devices. Application developers are released from complicated application partitioning and mobile users have the flexibility to isolate their computing environments and trigger remote execution in cloud on the fly.

The lofty goals of designing a computing-as-a-service mobile cloud system include a few aspects. First, the augmented virtual phone instances in cloud need to be

efficiently deployed and managed. Second, given that several virtual instances are co-hosted on the same physical machines, sophisticated scheduling algorithm is desired to optimize CPU scheduling to improve the physical hardware utilization efficiency. Third, an effective communication model is expected to maximize the gain of adopting cloud resources. In this thesis, we investigated these three building blocks of future mobile cloud computing system and proposed solutions for each building block.

In this dissertation, we propose a new computing model for mobile devices to access cloud resource. This thesis encompasses three main projects: Substrate, SBCO and AMPhone. In Substrate, we study the scale and deployment flexibility of VMs in shared data centers and introduce new techniques to manipulate intermediate VM states and construct new VMs on the fly. The SBCO project explores multi-core co-scheduling issues between consolidated virtual machines and proposes new vCPUs scheduling algorithms to improve the hardware utilization efficiency. The AMPhone project proposes a new computing model to augment mobile phone with the assistant of virtualized mobile phone clones in cloud. In our design, virtualized phone clones in cloud are deployed with our Substrate idea and managed with our SBCO approach. AMPhone elaborates the access scheme between mobile devices and the clones in cloud. Our motivation is to build a mobile cloud system in which cloud computing resources are offered as a service and virtual phone instances can seamlessly run on local phone natively or in cloud with augmented resources. In the remaining chapters, we first further introduce these projects and then discuss the organization of this dissertation.

1.3 Substrate: Agile VM Deployment

One major advantage of cloud computing is the capability of scale up on the fly depending on the needs of the applications. Such flexibility becomes possible mainly because the virtualization makes each component such as vCPU, memory size, storage space, network bandwidth highly configurable and the changes can take effect even without restarting a VM in the para-virtualization case. Many of today's applications, such as parallel computing, opportunistic job placement, call for agile instantiation and quick deployment of state-ful VM workers in the cloud. Other applications like short-term computing jobs even requires a new VM could be created in a real time manner. However, the traditional template based VM creation involves time consuming disk image copy, package installation, and system configuration, thus the total creation usually takes a few minutes, depending the total number of packages need to be deployed. This tedious process defeats the flexibility of the cloud computing and limits the performance of VM deployment.

Aside from the performance limitation of existing VM deployment, another major shortage of current deployment techniques is it does not preserve the intermediate result for the purpose of creating similar VM instances. Due to the intermediate changes including patches of kernel or applications are not preserved for VM creation, a new VM is often booted from a basic template with just enough OS(JeOS) and current applications and their dependency or configurations have to explicitly repeated in every new VM creation by duplicating virtual disk image, installing applications and configuring services.

In this thesis, we analyze the cost of each step in the process of VM deployment

and introduce the primitive of retrofitting VM deployment by using VM substrate to manage VMs in agile virtualized environment [93]. Our VM substrate-based VM shrinking and expansion management allows VM creating, reconfiguration in a way that is transparent to users and enables the instantiation of statefull VMs or VM clusters with sub-seconds latency. The VM pool design is capable of greatly reducing the latency of deploying new VMs and increasing the reusability of VM substrates. It incurs small overhead on the creation of a single or a cluster of VMs. Experiment results on the computation offloading from mobile devices show that the pool of VM substrates is able to provide instantaneous response to user request in an interactive job.

1.4 SBCO: Efficient SMP VM Co-Scheduling

Commodity OSES often use spin-locks for exclusive access to shared code or data. Such spin-locks require running processes to frequently acquire and release locks, while assuming only a short period of waiting time. In a virtualized environment, it is hard to keep the assumption when a vCPU is preempted while still holding a spin-lock and at the same time another sibling vCPU is still waiting for the spin-lock. Thus the sibling vCPU has to wait until the preempted vCPU to be rescheduled and releases the lock. Such issue is unique in multicore VM environments and often referred to as lock holder preemption(LHP). Depending on the usage of spin-lock for synchronization, the impact of LHP issue varies with applications.

In the thesis, we propose *SBCO*, a new scheduling scheme for performance optimization in virtualized SMP environment. *SBCO* first inherits the advantages of traditional co-scheduling such as minimizing synchronization latency and speedup the

communication between vCPUs. Meanwhile, it avoids the scheduling fragmentation and priority inversion issue because *SBCO* does not demand co-scheduling all the sibling vCPUs precisely at the same time. Instead, it coarsely adjusts the sibling vCPUs position in their respective run queues for balance purpose and facilitate sibling vCPUs to be scheduled coarsely at the same level. In other words, *SBCO* dynamically adjusts the affinity of vCPUs to avoid sibling vCPUs to exist in the same run queue. It also balances the sibling vCPUs in the different run queues. We implemented the prototype of *SBCO* based on CFS scheduler and conducted evaluations with KVM VM. Our experimental results show that *SBCO* brings more than 10% performance improvement for many applications.

1.5 AMPhone: Flexible Mobile Augmentation

In this thesis, we propose an innovative solution to run virtualized phone instances on both mobile devices and cloud. The virtualized instances on the mobile phone isolate users' different computing environments(including applications, user profiles, contacts and data) and sync with the augmented mobile phone clone in the cloud. The augmented copy has more computing power, more memory and disk space which contains the phone instances on mobile devices as a subset. Through impromptu launching augmented phone clones in the cloud, all time-consuming or resource hungry applications are shifted to run remotely and synchronize the results back to the phone instance on mobile devices. On the real phone instance, all the inputs, such as event from keyboard, touch screen are recorded and then send to the cloud clone where these these events get replayed by deterministic reply. The disk incremental updates of augmented phone instance is synchronized back to mobile device repeatedly. In

this communication model, we are able to significantly increase the computing power of mobile devices and reduce the energy consumption. Additionally, this model allows mobile users for quickly exchanging data in the cloud over high speed network.

1.6 Organization

The rest contents of the thesis are organized as follows:

Chapter 2 reviews previous work related to this thesis. We start with introducing some of the existing research work on promptly scale virtual machine deployment. Then we discuss the approaches of efficiently scheduling virtual CPUs in a consolidated environment. In the end of this section, We present a historical survey of mobile virtualization and compare a few different mobile computing models.

In Chapter 3, we first show that cost breakdown of tradition VM creation and then introduce the motivation of prompt VM deployment on the fly. We propose VM substrate concept as replacement of existing VM template and illustrate the processes of VM deployment from a VM substrate. The design of managing VM substrate with centralized pool as well as the implementation details are elaborated in the design and implementation section. We evaluate the effectiveness of the approach in terms of cost breakdown and show how efficient VM substrate can help online admission control.

In Chapter 4, we start with investigating the probability spin lock holder stack issue, which could significantly hurt the scheduling performance. Then we define the VM scheduling distance concept and explain its relationship to SMP VM scheduling performance with motivation examples. To address the issues, we propose a new way to efficiently balance sibling vCPU scheduling distance, namely SBCO and elaborate

its implementation details. Finally, we present experiments with multiple applications to show the effectiveness of SBCO comparing with other existing solutions.

In Chapter 5, we argue the case for mobile virtual machines as one promising solution for augmenting mobile devices limit. We first introduce the state-of-art mobile virtualization software and hardware. Then we discuss the challenges of shifting from server virtualization to mobile virtualization and the motivations combining mobile virtualization with cloud computing. We propose a new model, namely AMPPhone, to run a virtualized phone on both mobile device and cloud and synchronize the meta changes regularly. Finally, we elaborate the benefit of this model and introduce the implementation and evaluation details. We conclude by envisioning the future direction of mobile cloud computing.

Chapter 6 concludes this thesis with summaries of our proposals and the potential directions for future work.

Chapter 2: RELATED WORK

In this section, we introduce the state-of-art research efforts towards agile VM deployment, SMP VM scheduling and mobile augmentation with cloud computing.

2.1 Agile VM Deployment

VM templates are widely used to create new VMs in the majority of system virtualization platforms. Through preparing reusable templates, which are usually configured to include a standardized set of hardware and software configuration settings, the efficiency of deploying VM infrastructure could be significantly increased due to the fact that many repetitive installation and configuration tasks are avoided. A base VM template contains the essentials of server image so called Just-Enough-OS(JeOS) and the base template can be extended by installing software application(s) in order to generate new template. VM templates[64] can be either converted to virtual machines and powered on without deploying them. The conversion will either turn the original template into VMs which means the template doesn't exist anymore or clone the templates to VMs through replication which involves time consuming disk copy. Moreover, starting a new VM created from a VM template needs error prone booting process.

The Amazon Elastic Compute Cloud (EC2) [29] is a widely used cloud computing platform. EC2 allows users to create an Amazon Machine Image (AMI) containing their applications, libraries, data and associated configuration settings or use pre-configured, template images to get up and running immediately. Amazon's EC2 claims to instantiate multiple VMs in "minutes" is still not enough to meet requirement of some real time VM creation requests. RightScale [71] also provides scripts to

create and configure a basic VM from scratch. Although the installation and configuration are done automatically, it is often not applicable to on-demand VM creation due to the time consuming installation.

Some recent research work explores the idea of process fork to VM level where a running VM spawns child VMs that are clones of itself. The Potemkin project [91] realized a VM fork scheme that creates lightweight VMs from a static template locally within a single machine. Through aggressive memory sharing and COW techniques, Potemkin allows quick VM forking by deferring the duplication of memory pages until the contents of pages actually differ between VMs. It can support potentially hundreds of short-lived VMs on physical honeyfarm servers. However, Potemkin does not have the flexibility to create multiple VMs onto different hosts and does not offer runtime statefull cloning. Snowflock [43] extends the concept of VM fork in a distributed manner, enabling cloning a VM into multiple statefull replicas running in a cluster of machines. Snowflock leverages the same COW technique used by Potemkin and takes advantage of the high correlation of the children VM, providing a immutable image of the parent VM and a demand-paging mechanism to let children retrieve missing pages. Similar to process fork, VM fork is able to efficiently share parent's resources and swiftly create interim VM clones that run simultaneously in a real time manner. However, current VM fork implementations do not aim to deploy longstanding independent VMs. VM substrate is different from Potemkin or Snowflock in their purposes. Potemkin and Snowflock aim to provide on-demand virtual clusters with "identical" and "temporary" VM children forked from a single parent. VM substrate's objective is to preserve and restore customized user working space (VM's with different running states) with minimal cost. The VMs in question

are heterogeneous and not necessarily belong to the same user.

The idea of a pool structure is widely used in the design of computer systems. Most of early works focused on thread and process level pools [60, 14, 66, 53], or processor level pool [101]. The popular Apache web server [14] uses a thread pool to handle incoming request, but there is no resource reconfiguration for each thread. Iran Pyarali et al. [66] proposed an optimization to improve the quality of thread pools in real-time systems. They described the key patterns underlying common strategies for implementing RT-CORBA thread pools and evaluated each thread pool strategy from various aspects. In [53], Ling et al. characterized several system resource costs associated with thread pool size and analytically determined the optimal thread pool size to maximize the expected gain of using a thread and minimize the overhead of run-time memory allocation and deallocation while creating and destroying a thread. In [101], the authors proposed a class of scheduling algorithms based on a processor level pool which is used to organize and manage a large number of processors to improve performance.

2.2 SMP VM Co-Scheduling

While there are a number of research works to identify the performance overheads of virtualized execution [33, 46, 82, 67, 47, 55, 83], most of them focus on the overhead incurred by I/O operations or spinlock synchronization. The issue of preempting a parallel process which holds a lock has also been studied intensively in the past, see [13, 49, 83] for example. Virtualization makes the synchronization delay problem even more challenging due to LHP of a vCPU. If a vCPU is preempted out while holding a lock, then the lock waiter has to wait until the lock holder to be scheduled

again to release the lock. The LHP problem in a virtualized environment was first studied in [83]. In general, there are two approaches to address this issue: hardware assisted approaches and pure software scheduling solutions.

The hardware assisted approach detects lock holder with the low level hardware and assistant scheduler to schedule in and out the proper vCPUs dynamically to mitigate LHP problem. Modern processors provide architectural support for heuristically detecting contended spinlocks [8, 9]. For instance, PAUSE instruction is used by commodity OSes(e.g. Windows) in the spin lock for power efficiency consideration, therefore by identifying the execution of PAUSE instruction, the spin lock holder can also be detected [9]. In [94], the authors proposed a hardware assisted spin-lock mechanism to detect the cases in which a vCPU is not performing useful work and to suggest scheduler to preempt that vCPU to run a different, more productive vCPU. The heuristic lock-holder detection may cause frequent vCPU preemption. Moreover, this type of hardware assisted lock holder detection usually requires modifying guest OS, which is only possible with para-virtualization. This solution is not always feasible for guest OSes like Windows which is hard to instrument.

A typical software approach is co-scheduling, which was originally proposed to schedule concurrent threads simultaneously [65, 23, 84, 69]. Previous works [16, 89, 95] applied co-scheduling to SMP VMs to facilitate the vCPU communication and reduce application synchronization latency. They alleviate the LHP issue because all sibling vCPUs are scheduled simultaneously. However, classic co-scheduling algorithm has its inborn drawbacks such as CPU fragmentation, priority inversion and execution delay [48]. Moreover, co-scheduling is likely to cause more vCPU preemption for context switching, which is costly in virtualized environment.

To avoid the disadvantages of classic co-scheduling, an improved co-scheduling algorithm named balanced scheduling (*BAL*) was proposed in [81]. In stead of preventing LHP, *BAL* alleviates the effect of LHP issue by distributing sibling vCPUs to different pCPUs without forcing the vCPUs to be scheduled at the same time. It never delays execution of a vCPU to synchronize with other sibling vCPUs. It eliminates the drawbacks inherited from co-scheduling (CPU fragmentation, priority inversion and execution delay). Our *SBCO* inherits the advantages of traditional co-scheduling such as minimizing synchronization latency and speedup the communication between vCPUs. It coarsely re-adjusts the sibling vCPUs position in their run queues and facilitate sibling vCPUs to be scheduled coarsely at the same level. In another word, like the previously proposed *BAL* algorithm, *SBCO* dynamically adjusts the affinity of vCPUs and avoid sibling vCPUs from being dispatched into the same run queue. However, our *SBCO* is different from *BAL* because it balances the sibling vCPUs in different run queues by shorten their scheduling distance. This further reduces the synchronization latency in CPU over committed case. Our *SBCO* requires no hardware support and can be easily implemented.

VMware developed a few versions of co-scheduling solution for ESX server. The first version, referred to as strict co-scheduling, was designed for VMware ESX 2.x [89]. Later VMware created another relaxed co-scheduling (ESX 3.x) to moderate the severe CPU fragmentation in the older version. In the relaxed co-scheduling, all vCPU siblings are stopped but only the lagging vCPUs are started simultaneously when they goes out of synchronization. Such relaxed co-scheduling was further refined in ESX 4.x [87] stopping only advanced vCPUs, instead of all vCPUs. In all these co-scheduling approaches, scheduler tends to forcibly start or stop some vCPUs which

incurs significant context switching cost. Our SBCO balances sibling vCPUs and avoids arbitrary forcing vCPUs co-scheduling. Another hybrid co-scheduling framework was proposed in [95] to solve the CPU fragmentation issue, especially in CPU over-committed cases. It classified the VMs into concurrent VM when running concurrent workloads and co-scheduled all of its vCPUs. This co-schedule solution requires to determine the VM type manually. It is not applicable to scenarios where that knowledge is not available to system admins. In addition to these co-scheduling approaches, PACMan [72] provided some insights for performance aware VM consolidation. Matrix [22] proposed an approach to achieve predictable performance in cloud with machine leaning. Difference from these works that considers multiple contributing factors to performance, our approach focused on the LHP issue and attempted to alleviate its impact on performance and resource utilization. Gleaner [27] introduced an interesting idea to solve the blocked waiter wakeup(BWW) problem. Our work shares the same goal of reducing costly vCPU context switch. However, Gleaner consolidates short idle periods on multiple vCPUs into long idle periods on fewer cores, thus reducing the frequency that vCPU enter/exit idle loops. This approach may have limited improvement in heavy loaded cloud with CPU intensive applications. In those cases, vCPUs are busy most of time and they get rescheduled mainly due to running out of scheduling period other than entering idle loops. Our approach optimizes vCPU scheduling especially in over loaded cloud with high VM consolidation ratio.

There are other communication-aware CPU scheduling algorithms for collocated multi-tier applications [32] or NUMA scheduling [68]. Without considering LHP issue, this solution optimizes the default Xen scheduling to make VMM aware of

communication behavior of modern multi-tier applications. Besides these optimizing scheduling approaches, there were recent works focusing on optimizing the way of using spin-lock [34]. The authors proposed a new approach to manage the number of active threads to separate possible contentions with a load control mechanism. This contention isolation mechanism increases the efficiency of spin-lock and robustness of blocking. Though it does not require OS level modifications, the contention separation is application dependent and challenging in a dynamic virtualized environment. In contrast, our *SBCO* tends to optimize kernel level scheduler applicable to any type of workload. Note there are other works attempting to redesign the spin-lock. Raghavendra redesigned spin-lock and implemented para-virt spin-lock for KVM [40]. This approach does not prevent LHP. Instead, with the para-virt locks, the spin waiting time is reduced even when a lock holder is preempted. Similar to the hardware assisted approach, newly designed spin-lock requires guest VMs to be aware of the para-virt spin-lock. This may limit the usability of the spin-lock.

2.3 Flexible Mobile Augmentation

Recently, research on leveraging cloud to augment mobile computing has gained much attention. In this section, we present a few representative approaches towards this research direction, particularly, remote execution [74] and computation offloading [52]. Most of them focus on optimizing the power usage, augmenting performance or proposing a new computing model. We have analyzed them and compared them with our approach.

Remote execution was initially proposed to conserve the scarce resources on mobile devices in a few research work [15, 30, 75]. Remote execution requires moving com-

puting tasks from the mobile device to the server before task execution. The server performs the task and sends back the results to the mobile device. In [75], authors report that remote execution can save energy consumption if local execution is more expensive than remote processing cost. In recent years, remote execution has been adopted to boost the computing capability [70, 79, 24]. MAUI [70] proposed an mobile application code offloading system. In this system, all the methods or classes that could be executed remotely are identified by application developers and are offloaded to be executed in cloud. Though offloading is capable of boosting performance, such approach still has some limitations. The code that can be offloaded can not be the code for interacting low-level I/O devices or internal resources of mobile devices.

Cloudlet [79] proposed another infrastructure for remote execution. In a cloudlet system, mobile devices provide input to launch VMs in the cloudlet. These VMs provides services according to the input and send the result back. Mobile devices are connected to a cloudlets via high speed wireless network. The main motivation of this proposal is to provide interactive communication between mobile device and cloudlet. This model is essentially to make mobile devices as a terminal to access local resources instead of directly using the resources available on mobile devices and it is also limited to certain type of services that cloudlet can offer. Clonecloud [24] further explores the feasibility of remote execution by proposing moving the whole or part of the execution of resource expensive applications to smart phone clones in more powerful computing infrastructure. The augmented phone clone is synchronized with the mobile device via whole system replication which is tradeoff due to the synchronization cost. The partial remote execution model in the proposal also requires separating resource expensive code. Based on MAUI and CloneCloud, ThinkAir [39]

introduces a framework for code offloading with parallel processing capability for mobile applications. It targets a commercial cloud cases with multiple mobile users and considers the elasticity and scalability of the cloud for the dynamic demands of customers. Unlike the existing remote execution approaches, we propose running a augmented phone clone of a real phone in cloud. We focus on whole system replication but incremental synchronization. The clones are running in a virtual cluster which has the same CPU architecture with mobile devices. Thus there is no emulation or code isolation. Our main target is saving power consumption and agumenting applications' performance.

Besides code offloading type of remote execution, thin-client [35, 45] offers another option for remote execution of some applications. The initial thin client design was proposed to reduce user delay experience due to remote exectuion. The remote execution results are synced back to mobile devices with high speed network and the execution process is transparent to mobile users. Applications like web browsing can benefit from thin-client approach because the CPU intensive webpage parsing is done remotely. Unlike this thin-client approach, our work involves both virtualization on mobile phone and virtualization in cloud. The virtualization on mobile devices give users more flexibility of running multiple environment concurrently.

Towards the direction of enhancing mobile devices' computing capability, recent research work [10, 76] proposed to isolate the computing intensive code and data and them outside of mobile devices. Only the lightweight and less intensive code and data are saved locally on mobile devices. Such isolation reduces the overhead of identifying, partitioning and migrating resource intensive tasks and increases the reusability of those isolated tasks. Although this isolation alleviates the overhead, it is

application dependent because programmers' involvement is needed for the isolation. Our proposal release application developer from this complicated isolation.

It has been proven that augmenting mobile devices with cloud can increase their computing capabilities and conserve energy. Unlike most of the existing work, our proposal has two main differences. First, the operating system on mobile devices is virtualized and more than one instance can run concurrently. Such virtualization on mobile devices gives users the flexibility to isolate different environments. Second, the augmented clones are also running as VMs on the same CPU architecture as mobile devices and there is no emulation. Application developers will have access to a virtualized mobile platform that is very accurate to the real device.

Chapter 3: AGILE VIRTUAL MACHINE DEPLOYMENT

In this chapter we describe how we tackle the problem of agile virtual machine deployment and virtual machine resource management.

3.1 Introduction

Cloud computing in its original form offers virtualized resources, and infrastructure in general, as a service over the Internet. A key requirement is resource provisioning on-demand in a real-time manner. In the model of infrastructure-as-a-service, applications are often run in virtual machines (VMs) and their performance relies on effective management of the VMs in the whole life-cycle from creation, deployment, execution, to termination. Because of the nature of on-demand computing, VM startup latency is a crucial performance factor in application responsiveness, in particular for those that interactive, impromptu, and short-lived computing [44].

An example of such applications is server-based computing (SBC) [51], in which resource-constrained client applications offload compute- or data-intensive tasks to VMs running in a data center, e.g., through computation offloading or wrapping mobile OS to VMs running in the cloud can significantly extend the computing capability of mobile devices as well as save the scarce battery resource. [24]. In such case, the VMs may need to be created and deployed on the fly during the execution time of the applications. Another example is virtual desktop infrastructure (VDI) [85], in which clients would launch their VMs associated with their personalized working environments and data on a remote client device upon request. In addition, in virtualized parallel computing, the size of a VM cluster varies with the workload which requires new VMs worker can be created instantaneously. Startup latency is

pivotal to the success of all these cloud computing usage cases.

VM creation from scratch requires to create a virtual hard drive image, configure virtualized resources, install OS and initialize application services. This process would take tens of minutes. To reduce the startup latency, in practice, public IaaS providers like Amazon Web Services provide users an option to create VMs from template. A VM template [86, 64] is a reusable image created from a clean VM and stored in disk as a file. Although a VM can be created by booting from a template in tens of second, the template become non-reusable by others. VM cloning from a template would retain the reusability of the template but at the cost of expensive disk copy of large image files. In either approach, there is no time-efficient way to create multiple VMs simultaneously from the sample template, although such parallel deployment is crucial to parallel computing and server clustering.

There were recent studies on reducing the startup latency and supporting parallel deployment; see Potemkin [91] and Snowflock [43] for examples. Potemkin proposed a delta virtualization technique for flash VM cloning. It relies a copy-on-write optimization technique to have multiple VMs share memory pages as much as possible. Snowflock proposed a process-fork like API to fork VMs for parallel processing during the execution of a program. The VMs created inherit the software stack from their parent VMs and can not exist without the presence of their parents.

In this work, we propose an abstraction of VM substrate as an alternative to VM template for rapid deployment and parallel deployment of VMs. VMs created from substrates have the same life cycle as template-based VMs and the VMs are of independent by origin and can be deployed across different physical hosts. Unlike templates that are stateless and stored in disk as an image file, substrates is a generic

Template Size	2G	5G	10G	20G
cp(local disk)	36.06s	58.75s	547.45s	1228.69s
cp(nfs)	46.16s	78.21s	640.28s	1412.42s
scp	43.31s	114.66s	749.97s	1589.35s
dd(single disk)	3.07s	45.55s	195.71s	515.17s

Table 3.1: Cost of creating VM from templates.

VM instance in miniature that docked in memory of a designated machine in an inactive state. They can be present with or without application footprints and ready to be powered on upon request. Creation of VMs from substrates saves time from time-consuming disk-based booting and deployment. The substrate mechanism leverages an array of techniques, including VM miniaturization, generalization, clone and migration, page copy-on-write, and on-the-fly resource configuration, to save memory space, generalize substrate usages, and resolve resource configuration conflicts on VMs to be created. The mechanism facilitates parallel VM deployment via multicast.

We have implemented a prototype on a Xen/Linux server cluster and tested the system in two scenarios: on-demand deployment of VMs for cloud-assisted gaming and parallel deployment of heterogeneous VM clusters like LAMP (Linux/Apache/MySQL/PHP). Experimental results showed the mechanism capable of creating VMs in subsecond, while retaining the flexibility of VM resource configuration. The experiment results also show that the substrate mechanism makes it possible to deploy a VM cluster in a few second or a speedup of more than 50 times in comparison with default VM deployment from template.

3.2 Background

Deployment of a VM in a data center involves a number of steps: (1) VM creation with virtual hard disk; (2) Installation of OS images and applications; (3) Deployment with configuration (networking, etc) on selected host/cluster; (4) VM startup.

New VMs can be created either from scratch or from template. As the process of VM creation from scratch takes tens of minutes, it is rarely used in cloud. On the other hand, deploying a VM from templates, which removes the process of OS and software installation, is widely used in practice. VM creation from templates involves two steps: (1) create a copy of the template's virtual disk image and (2) customize the VM configuration as needed. Configuration customization includes parameter settings for boot option, host name and network. VMs can be created from templates through either cloning or conversion. VM templates are usually created for a specific purpose such as a web server or a database server. Once booted, the VM which originates from a template can be further extended by deploying more applications or run-time libraries. In the following, we first discuss the cost of VM creation and then examine the state transition of a VM. Next, we present the challenges of fast VM deployment.

3.2.1 Cost of VM Creation

The cost of template-based VM creation comes from different sources. First, depending on the storage environment and VM template image size, the cost of VM disk duplication varies. In order to support VM live migration [25], VM disk images are usually stored in centralized storage servers. NFS and iSCSI are two popular choices for the deployment of VM virtual disks. In either case, the duplication of the template's disk image is necessary for a new VM creation. Table Table 3.1 shows the

cost of disk duplication with different disk sizes and different methods. Regardless the underlying storage organization and duplication methods, the cost increases significantly with the VM disk size. A 5GB VM disk requires more than one minute to be copied. The latency incurred by disk duplication is not acceptable to interactive applications. Besides, according to the table, to clone a new VM from a template on remote host (*scp*) takes tens of seconds or even a few minutes, consuming a significant amount of network bandwidth in the data center. Note that although create a blank disk image on local disks (*dd*) takes less time, but deploying root filesystem takes even more time than directly duplicate a disk VM with root filesystem as a whole. Second, the booting process of a VM includes booting the kernel and starting default services. Kernel booting usually takes sub-seconds while starting different services is both error-prone and costly. The general purpose OS installation activates many services by default. RightScale [71] templates and Oracle VM templates [64] disable most of the application unrelated services to minimize the cost. Third, traditional JeOS templates are usually extended by installing more applications to generate new application specific templates. The cost of maintaining various VM templates increases with the diversity of application oriented templates. All these costs together makes template based VM creation impractical for interactive applications.

3.2.2 VM State Transition

Starting from a template, a VM experiences multiple states in its life cycle. Figure Figure 3.1 shows state transition diagram for a VM. Each VM is initially halted after being created from scratch or cloned from templates. Although each halted VM is a static instance only consuming disk space, it still can be edited or customized by installing new applications or changing the associated configuration. A VM is

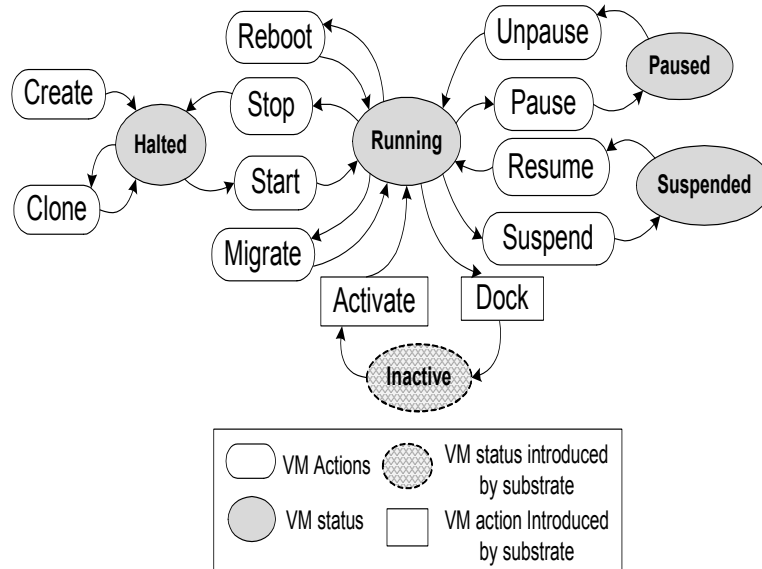


Figure 3.1: VM State Transition.

changed to a running state when it is started and A VM can be paused or suspended on local host or migrated to another host. We added one additional state and two new actions to the conventional VM state diagram [64]. A new substrate is generated from a running VM through docking. Docking can be done by converting or checkpointing. Converting puts the running VM to an inactive state, while checkpointing keeps the VM running. The tradeoff between these two solutions are discussed in substrate design section. Note that an inactive state is different from a halted state. An inactive VM consumes memory and maintains running status, but a halted VM only consumes disk space.

3.2.3 Challenges of Rapid VM Deployment

Rapid VM deployment calls for minimal costs in each step of VM creation. However, as discussed above, virtual disk image duplication is time-consuming. It leads to a large startup latency. Moreover, if multiple VMs need to be created at the same

time, disk duplication is the key impediment to fast VM deployment. In addition, the automatic resource reconfiguration of new VMs is also challenging, especially in a heterogeneous virtualized cluster of VMs with interactive applications.

Stateless VM creation has limited usage cases due to the fact that it creates brand new VM every time without preserving runtime environment or intermediate result. A brand new VM with necessary applications pre-installed is how the general VM template is used. This is insufficient for many of the cloud applications like parallel computing or mobile computation offloading. Thus the fast creation of statefull VMs is necessary.

Rapid VM deployment also requires that the creating process should be transparent to users and applications. Because creating a new VM always takes time, in the cases of user interactive applications or other request-driven VM creation, startup latency caused by creating a new VM must be small enough so as to make the creating process transparent to application. If the cost of creating process is negligible, from applications' perspective, VMs are always ready for use.

3.3 Design of VM Substrate

Modern applications and libraries consume a considerable amount of disk space, which makes the size of templates usually large. To address these limitations, a few questions need to be answered. First, can image file be stored in memory instead of disk? Although, solid-state-disk(SSD) attempts to increase the efficiency of data transfer between disk and memory, it is still not fast enough to meet the requirement of duplicating disk image on demand. Moreover, the size of traditional templates can easily go beyond the limitation of the memory of a server or a common SSD. Thus it is

impractical to maintain templates in memory and only a limited number of templates can be saved on SSD. Second, is it possible to avoid the booting process while still maintaining previous running states when starting a VM? An AMI [29] or oracle VM contains a minimal Linux installation with only essential Linux services, leaving the installation of additional applications to package management tools. Thus the images are much smaller than default Linux OS installation. However, it is a brand new OS with only a limited number of services installed. Third, is it possible to deploy a VM in a real-time manner? Real-time VM deployment allows VMs to be created on demand and only be activated when in use. In the remaining section, we elaborate the design of VM substrate and compare VM substrate with alternative approaches.

3.3.1 VM Substrate and Pool

The design of VM substrate aims to leverage existing virtualization techniques to provide an agile cloud computing environment which allows users to create VMs or VM clusters on demand. A VM substrate is a static reusable instance that can be duplicated or reactivated for later use. VM substrates are categorized into three types. Public substrates contain minimal clean JeOS and generic configuration. Restricted substrates are the extensions of public substrates with specific applications and runtime environment. Alternatively, private substrates include users' personal data which can only be reused under strict sharing policy. These types of substrates are designed for different use cases, but they follow the same docking and reactivating process.

Saving the running states of VMs into in-memory VM substrates has many advantages over having VMs always run in full capacity. If a VM in full capacity is paused or suspended to the local machine, the resulted memory footprint which contains the VM's running state is usually quite large, in proportion to the VM's original

capacity. If the saved state is stored in local machine's memory, the restarting of the paused/suspended VM is instant but at a cost of wasted memory resources which can be otherwise used by other running VMs. If the state is saved on local hard disk, the time required to resume the VM is unacceptable. For example, it takes approximately 40 second to restore a VM with 2 GB memory from a 7200RPM SATA disk. In VM substrates, we first trim the VM to its minimal capacity (minimal CPU and memory, detached block and network devices) that preserve essential running states, and then temporarily dock the VM to memory other than disk. After the final compression, the resulted memory footprint which usually in a size of tens of megabytes is transferred and consolidated to a dedicated substrate pool. Upon resuming, the corresponding VM's substrate is activated by expanding to its real capacity. The restoration latency is comparable to the local in-memory restore but with a much less memory cost on each local host.

A substrate pool is a centralized repository where all the substrates are maintained. Unlike traditional VM template pool, The substrate pool stays mainly in memory and the backup substrates are stored on disk. The size of a substrate pool is dynamically reconfigurable without affecting the existing substrates. Our preliminary experiment results show that a substrate with minimal programming environment can be as small as 16MB. With the similar substrate, we successfully hosted several hundreds of substrates on a physical machine with a 4GB memory. However, the sizes of the substrates depend on the running status of the hosted application. In order to maintain a statefull substrate with manageable cost, we aim to embed only necessary data into a substrate.

VM substrate is proposed to be an alternative effective VM administration solu-

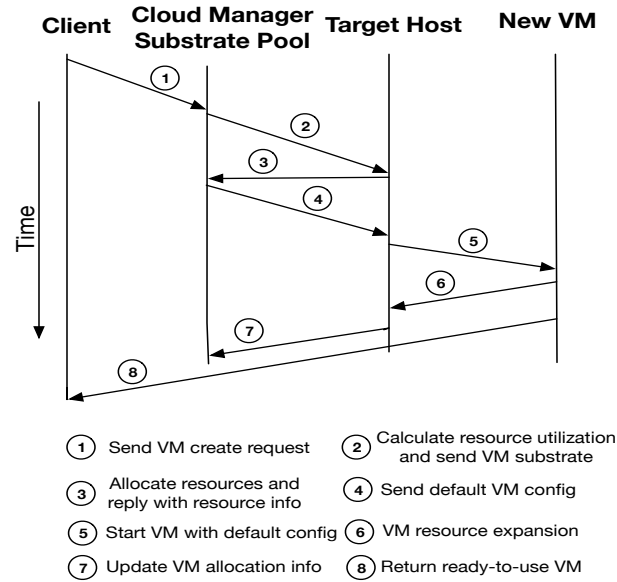


Figure 3.2: Create VM from substrate.

tion not only applicable to instant parallel workers creation, but also applicable to standalone VM deployment, also taking the reusability and scalability into consideration. Different from VM Descriptors proposed by Snowflake [43], VM substrate doesn't have heavy dependency on any parent VM and has many varieties. VM Descriptors contain only the minimal critical metadata needed to start execution and use *Memory-On-Demand* mechanism lazily fetch portions of VM state over network as it is accessed. In contrast, VM substrates are static VM abstraction resides in a pool in memory. Activation, resource expansion and remapping are the typical three steps to create a new VM from a substrate. In theory, it is possible to maintain a pool of template parent VM and then fork child VMs on demand. However, this solution can hardly get rid of the limitation of dependency and hard to meet the requirement of VM creation for long standing services. Moreover, due to the size of the parent

VM, the cost of maintaining template parent VMs is much higher than maintaining a substrate pool. In addition, if the application is CPU intensive and requires minimal updates to disk or the intermediate results can be discarded, an alternative way to VM fork is to start multiple VMs on different hosts with the same disk image located on a centralized server. But this solution has very limited usage cases.

The abstraction of VM substrate introduces two VM state transfer actions in the life-cycle of a VM which are docking and activating. A VM substrate is constructed by docking a running VM maintaining applications' running status. There are two ways of docking: intrusive converting of a running VM and live checkpointing of a VM. In contrast, VM substrate activating includes dispatching substrate, launching substrate and reconfiguring substrate's resources. A new VM is created after the activating process.

3.3.2 VM Clone From Substrate

We employ four steps to address the challenges in on-the-fly VM creation. First, VM miniaturization and generalization. Before generating new VM substrate, the parent VM is shrunk to a miniature state. A VM substrate has minimal memory footprint, single vCPU core, detached network interface and reference to virtual disk. Since the memory size is a major factor of the final size of a VM substrate, the memory size needs to be shrunk to the greatest degree through either intrusive shrinking or live checkpointing. In either case, the data in the system cache is synchronized to disk first. Through predictive calculation, we reconfigure VM's memory to a size that only contains data necessary for the restoration. VM configuration generalization assures the VM specific configuration of public or restricted VM. Configurations such as host name, networking parameters are reset to the default value. The resources of

a private VM substrate is minimized while still maintaining its original configuration. Second, raw VM substrate is generated right after the VM's resource shrinking. A snapshot of the minimal running VM is created and stored in local memory. Third, raw substrates are compressed to be the final VM substrates before they are moved to a substrate pool. Compression reduces the substrates to a size as small as tens of megabytes which can be transferred over WAN. Fourth, the minimal VM substrate on local memory is transferred to a centralized pool. Figure Figure 3.3 illustrates the steps of docking a running VM to a substrate.

When a substrate is selected to create a new VM, as shown in Figure Figure 3.2, it is duplicated to other physical hosts simultaneously via multicast. Each physical host then decompresses the VM substrate and activates it from memory. Through reconfiguration, newly created VMs on each host will be allocated more memory and vCPU resources depending on application needs. New network interface with predefined parameters is attached to the VM and the configuration takes effect immediately. Depending on the type of a substrate, root disk is remapped and user's personal disk partitions can be attached to the VM.

3.3.3 VM Substrate Generation

Converting a VM to a substrate starts with reconfiguring a running VM's resource to minimal memory footprint and vCPU number, detaching the network card and saving the disk states. The initial VM from which a substrate is constructed can be a VM template or any VM with applications running. Intrusive conversion can be initiated in the application level by administrators whenever the VM has no scheduled work and is ready to be docked.

A VM substrate can also be created through live checkpointing in system level.

VM checkpointing has been widely used for various purpose like high availability [26], VM migration [25, 58, 21, 92], fault-tolerant [57] or debugging [36]. We also leverage checkpointing to create VM substrates without interrupting the running services. Most of existing VM level checkpointing techniques tend to save the entire running states (*cpu, memory, disk*) in a core dump where the resulted checkpoint size is the VM's memory size, and the checkpointing time is closely related to memory page dirty rate. We employ two techniques to ensure that a VM can be correctly restored from a substrate and the size of the resulted substrate is minimized. First is selective memory checkpointing, through which only reusable memory pages are saved to substrates, discarding the reconstructable or zero pages. Selective checkpointing memory is able to reduce the size of raw substrates considerably. Second is the generalization of VM configuration, which set all VM specific resource identifiers like *vmid* or *uuid* to default values in a VM substrate. By using these two techniques, a checkpointing substrate of existing running VM instance can be created any time without conflicting with the original VM.

Compared with intrusive conversion, live checkpointing is able to create VM substrate without interrupting user applications, but it requires the modification of the VMM for selective memory dumping. In contrast, application level substrate conversion is independent on the underlying VMM. It only requires that virtual hardware resources of a guest VM can be configured dynamically without a restart.

3.3.4 VM Fork

We note that VM fork has been recently proved to be an efficient way to clone a parent VM to multiple copies swiftly [43, 91]. Similar to process level fork, VM fork allows a child VM to inherit all the states originated from its parent VM prior

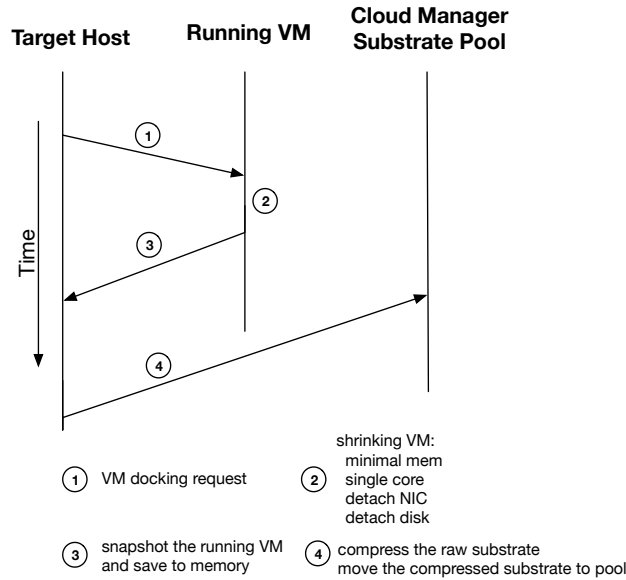


Figure 3.3: VM dock to substrate.

to forking, enabling creating statefull computing instance rapidly. However, different from process fork, VM fork is capable of creating VM clones across a set of physical hosts. It can also work in a parallel manner where a single API call launches multiple VMs. Each child VM has its own independent copy of resources and runs independently from the parent VM. Once forked, and the changes made to each cloned VM are maintained separately. We analyze the advantages and the disadvantages of VM fork and compare it with VM dock and reactivate in the remaining of this section.

VM fork is capable of creating transient VMs whose virtual resources are discarded once they exit. The intermediate states or values generated by the applications in a child VM are lost unless being explicitly synchronized to the parent VM. Due to the characteristic of a fork operation, VM fork has a few limitations. First, VM fork is applicable to computation intensive applications with limited or disposable inter-

mediate results. Existing VM fork leverages disk Copy-On-Write(COW) techniques to offer each child VM a COW slice of disk and all the disk updates or intermediate values are preserved on the COW disk. The child VMs share the running environment of the parent VM and the coordination between the parent and the children is mainly limited to computation. In the case of IO intensive applications, each child VM needs to make changes to their own disks which are actually COW slices. When the tasks in children VMs finish the updates on each child may need to be synchronized back to the parent. The integration of the updated data to the base disk incurs significant cost. It is challenging to achieve consistent synchronization once several VMs changed the same data. Second, sharing the same base disk partition between parent and children VMs limits the scalability of VM multiplexing. With IO intensive applications, the disk bandwidth of the base partition can easily become the performance bottleneck. Although *multicast* can be used to render memory pages concurrently to all the children VMs and memory page prefetching can possibly speed up on-demand paging, VMMs like Xen only grants the privileged domain direct access to the devices and does not allow the guest domains to access them directly [62, 32]. If the number of child VMs that request missing pages is large, the parent VM would receive a considerably amount of page requests from network interface. The parent VM can possibly become a hot-spot.

Third, current VM fork implementation remains at application level focusing on parallel applications which need to re-spawn additional temporary workers. However, VM fork is not ideally suitable for deploying longstanding independent VMs at cloud administration level. Server applications such as web hosting and database warehousing usually run in loose coupled virtual clusters with minimal correlation.

Such applications often require persistent data storage for each virtual node. Another drawback of the VM fork mechanism is its inability to create a heterogeneous VM cluster at a time. The VM substrate approach proposed here tries to create a cluster of heterogeneous VMs in a real time manner.

3.4 Implementation

We have implemented our VM substrate pool mechanism on the Xen platform. Xen is capable of running two leading approaches for virtualization: para-virtualization(PV) and full virtualization(FV). FV is designed to provide total abstraction of the underlying physical system, in which guest OS or applications are not aware of the virtualized environment. However, it incurs much performance overhead and can not be reconfigured on the fly without reboot of the VM. In contrast, PV presents each VM an abstraction of the hardware and requires modification of OS, allowing near-native performance. The memory size and the number of vCPUs of a PV guest VM can be reconfigured without restarting the VM. Thus, we select PV VMs in our prototype implementation. Our implementation includes modifications to the hypervisor, the `libxc` library, and the `xend` management daemon. In the remaining of this section, we elaborate the implementation details and compare them with alternative approaches. We also present micro-benchmark results to show the feasibility and effectiveness of the VM substrate.

3.4.1 Resource Shrinking and Expanding

vCPU: vCPUs are what the guest sees as CPUs on which the guest OS schedules applications processes or thread. The final size of a VM substrate is not affected by the number of vCPU configured in a VM. In order to make each substrate be more

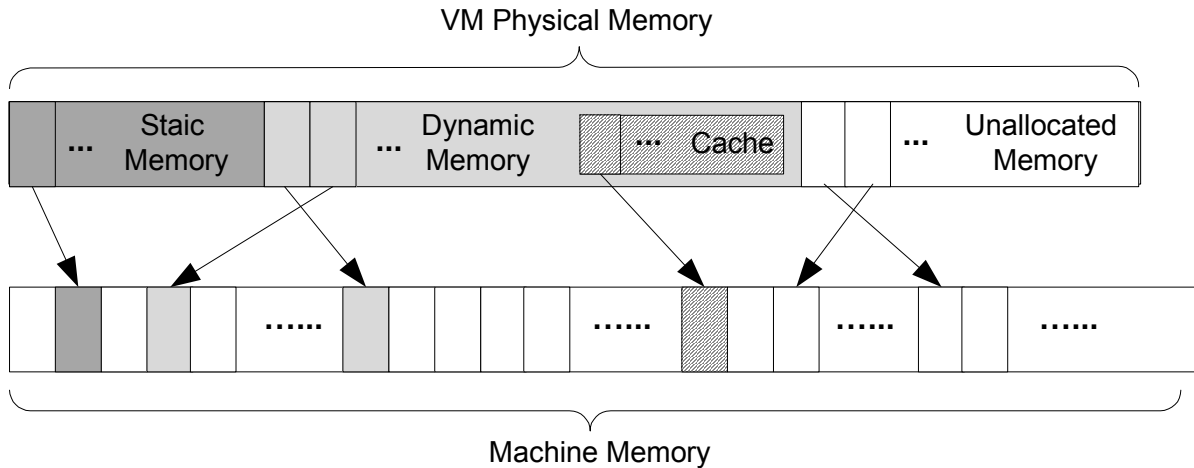


Figure 3.4: VM's memory footprint.

generic and with minimal resources, each VM substrate has an default configuration of a single vCPU core. In practice, vCPUs are usually pinned to specific physical CPUs for predictable performance. VM substrate is designed to be a generic mechanism that does not assume any physical host information. Thus, CPU affinity information is not maintained in the substrate. In a heterogeneous cluster, a VM substrate with a single vCPU is able to be deployed on any physical machine. Since Xen VMM does not allow the actual vCPU number to exceed the maximal number of vCPU specified in the guest's configuration file, we set the default maximal number of vCPU to be the total number of physical CPU cores for each substrate. Any newly created VM initially has single CPU core by default. More vCPUs can be allocated at a step of one vCPU.

Memory:

Xen VMM is responsible for managing the allocation of physical memory to guest domains and maintaining a triple indirection model (*virtual memory, pseudo physical memory and machine memory*). Each VM runs in an illusory flat, continuous address

space. Xen reserves the top 64M of the virtual address space for every domain. The remaining physical memory is available for allocation at a granularity of one physical page. Xen maintains a globally readable mapping table between PFN(Pseudo-physical Frame Number) and MFN(Machine Frame Number). The OS running in a VM maintains the mapping between virtual memory and pseudo physical memory. As shown in Figure Figure 3.4, each VM's physical memory is part of the machine memory and can be divided to several parts including used pages and unallocated free memory. The used pages can be further divided into static memory pages and dynamic memory pages. The later one also includes disk cache. Note that although the used pages are not available for reallocation, it is still possible that some of those pages are zero pages either because they are set to zero by programs or they are used as heap initialized by compiler. Traditional VM save `xen save` writes the VM's entire memory including zero pages, cache pages and free pages to a checkpoint file. Including free and zero pages in the checkpoint file is likely to be a waste because those pages store no information of the checkpointed states. In order to minimize the size of a VM substrate, we only keep the reusable and minimal memory footprint while still maintaining the integrity of a VM's state.

A VM substrate which excludes free pages does not harm the correctness of VM when it is relaunched because those free pages can be easily reconstructed by manipulating the mapping table of MFN and PFN. Zero pages are still included in a VM substrate for the following reasons: First, there is no more efficient way to extract zero pages other than doing a bit by bit comparison. The cost rises as the size of VM memory increases. Second, each VM substrate is compressed before going to a substrate pool, the compression algorithm is capable of compressing the zero pages

with a large compression ratio which reduces the size of the substrates considerably. Note that disk cache is used for performance optimization where recently accessed data can be retrieved from memory without incurring disk IO. Before creating a public or a restricted VM substrate, disk cached pages are synchronized to the disk which yields more free pages and the final substrate size can be further reduced.

Most of existing Linux distributions enable many optional services by default even for a base installation. Rightscale[71] uses bash scripts to disable those optional services before building a template. In addition to kicking off disk cache pages, we also release part of the memory occupied by killing user applications that are not relevant to the main purpose of the substrate. For example, in a substrate dedicated for web hosting applications, optional services like *sendmail*, *nfs* can be removed. We customize the application level services before docking a VM.

Memory ballooning is used by VMMs like Xen to achieve memory over-commitment. It provides the ability for the sum of the physical memory allocated to all active domains to exceed the total actually physically available memory on the system. Recent dynamic memory balancing work [100] proposed mathematical models to forecast memory needs and dynamically adjust the memory for VMs. The objective of these two memory adjustment approaches is to improve memory utilization. The later one also considers applications' throughput and performance. It is possible to instrument Xen to track memory accesses with each VM through the use of shadow page table. Shadow page tables are enabled during Xen's VM migration to determine which pages are dirtied during the migration. However, trapping each memory access results in a significant application slowdown and is only acceptable during migration [25, 77].

After a new VM is created from a VM substrate, it will start running at the initial

state with minimal memory. It later expands to a larger size according to the setting in the configuration file. Each VM has a maximum and current memory size. Current memory size can be adjusted up to the maximum size. We configure the maximum memory of each substrate to be the physical memory size. The total memory size is extended dynamically. We implement an application level memory shrinking mechanism which is used to convert a VM to a substrate based on simple speculation in our prototype. We use the Linux /proc interface (in particular /proc/meminfo) to analyze the memory usage. Before docking a VM, we first kick all the cached data back to disk and consider the remaining memory size being actively used. Then we determine the minimal amount of memory the VM needs by adding a safe margin preventing Out-of-Memory crashes when the VM is restarted from substrate. The VM is set to the resulted memory size. The memory footprint of a guest VM will directly influence the final size of the VM substrate. The effect will be evaluated at the end of this subsection.

Network: The privileged domain in Xen VMM implements the network interface driver and all other guest domains access the driver via virtual device abstractions. Each domain is attached one or more virtual interfaces. Due to the fact that virtual interfaces are not necessary for booting a VM, their configurations can be postponed until rest of the guest OS ready to work. Conventional migration keeps network connection status by maintaining all protocol states and keeping IP addresses and MAC addresses in a record. Existing solutions used to manage network configuration during migration are to generate an unsolicited ARP reply from the migrated host, which lets the switch and other hosts know that the MAC is connected a new port [25]. However, even if the switch is configured not to block ARP broadcast, conflicts still

exist if multiple VMs are created from the same substrate because all the network configurations of the new VMs are originated from the same substrate. In order to avoid the conflicts, We detached the network interface before docking a VM and VMs created from substrates do not have network interfaces initially.

The network parameters are configured when a new network interface is attached to a VM. In our prototype implementation, we also developed a mechanism to isolate the network in order to prevent interference between unrelated VMs. First, the networking mode (NAT, bridge or routing) can be dynamically configured with an interface in a physical host. Besides, the IP, MAC addresses and even the network mode can be determined within a physical host and transferred to guests as parameters. We implemented guest network configuration mechanism based on `Xenstore` to provide agile and immediate configurations. Depending on the purpose of newly created VMs. Especially when a virtual cluster is created, they are deployed with private network addresses and only guests within the same subnet are visible to each other.

A VM substrate is the snapshot of an original VM, and the memory and process running status are preserved in the substrate. This may result in some conflicts if new VMs are created based on one VM substrate because they share the same running environment. It is possible that multiple processes in different VMs may need to connect to the same socket or open the same file. In our prototype implementation, docking VM can be done at administrative level when one phase of computation is finished or before the application starts to run. Another solution is to create a substrate directly from a running template.

Disk: Disk image files are commonly used as virtual disks by guest VMs. Because

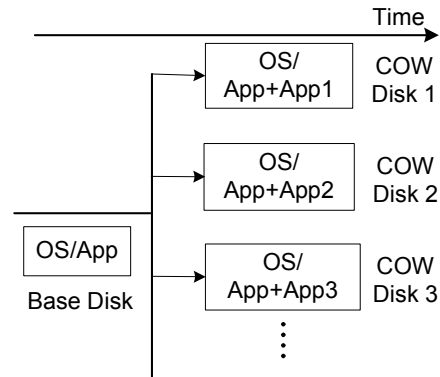


Figure 3.5: Vertical COW.

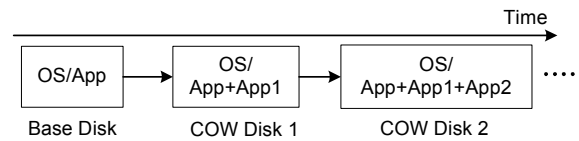


Figure 3.6: Linear COW.

the disk image files, which are usually in a size of tens of Gigabytes, stores the application specific data, costly disk duplication is often unavoidable if new VMs are to be created. Existing template-based VM creation simply distributes the virtual disk image in a copy-and-paste manner to reconstruct the same VM without reinstalling OS or applications. Thus any two VMs from the same template are independent from each other, guaranteeing the isolation of VMs. However, the time spent on copying virtual disks is unacceptable provided that the disk size is usually large. Disk copy-on-write is often used to avoid unnecessary disk space waste. Multiple COW slices can share the same read-only base image file and all the updates are directed to those COW slices. Wide-area VM migration used disk COW to transfer VM disk state over low bandwidth and high-latency links [77, 42]. To reduce the startup latency of new VMs, disk COW is also used recently by Snowflock [43] and Potemkin [91] to

generate temporary disk slices for newly created VMs.

There are two different types of disk COW. First, a *blocktap* driver combined with a *qcow* slice, which is supported by Xen VMM. Second, LVM supports creating writable snapshots of logical volumes quickly and each snapshot can be used as a COW disk slice by guest VMs. However, both of these two approaches have their limitations. Traditional *qcow* based COW has a limit on the total number of slices created and also has to make the tradeoff between the size of the COW disk and the depth of the COW disk hierarchy. Deeper hierarchy leads to bigger image files. Figure Figure 3.5 and Figure Figure 3.6 illustrate two typical ways to create a COW disk partition. The linear COW approach in Figure Figure 3.6 applies incremental COW slices onto existing disk partitions. The existing disk partition can be an initial base partition or a partition already having COW slices on it. The vertical hierarchy as shown in Figure Figure 3.5 dedicates a VM to a single purpose with fewer applications installed, thus it is able to limit the resulted partition size to a certain extent. In order to avoid the high The root COW disk is the initial image file and usually installed with the JeOS, then multiple child COW disks are created afterwards with each taking the previously created root COW as its parent and install with different kind of application. Due to the IO scheduling of virtualized disks, more COW slices result in higher dependency and the more degradation of the performance in either mode. Moreover, it is very challenging to merge multiple COW slices to the base image because the order of updating disk file is usually not preserved. On the other hand, LVM snapshots usually appear as a physical partition and requires using tools like ATA over ethernet(AoE) or iSCSI[56] to export COW slices when VMs need to be deployed across multiple hosts. Each new slice requires an update to the running

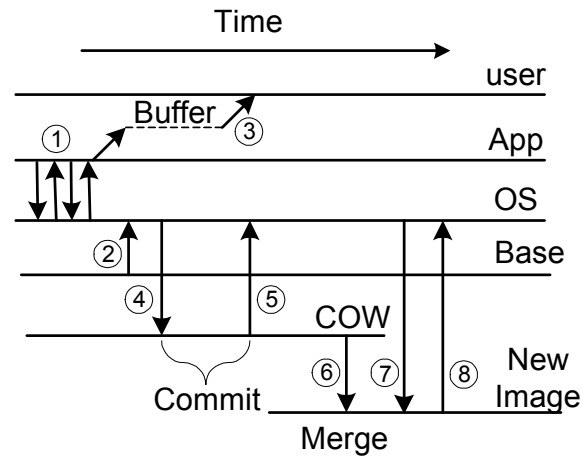


Figure 3.7: IO sync.

AoE or iSCSI service to export a new disk partition. In addition, only recent LVM version supports merging a COW back to the base and it also needs to use the latest Linux kernel. In conclusion, disk COW slice is only applicable to temporary VM creation.

Xen disk block device supports split driver model and the VMM provides a mechanism for device discovery and data movement between domains. The device drivers are split across Domain 0 and guest domains which are also called back-end and front-end respectively. Domain 0 is responsible for supporting hardware, running back-end devices drivers and providing the administrative interface to Xen. This VM disk model allows that a VM's disk can be reconfigured. We leverage COW techniques for substrate-based VM deployment with some modifications to the existing COW mechanism. The objective of real time VM creation are two folds. First, in the long run, VM substrate-based VM creation should guarantee the correctness and should generate consistent application result compared to the VMs created from templates. Second, from the users' perspective, a VM can be created on the fly in a real time

manner with small latency. Inspired by [59], We create a temporary COW slices and remap it to a newly created VM from substrate, giving users near realtime responses to the VM creation requests. The temporary COW slices work as the root partitions in order to speed up the booting process. At the same time, we duplicate the base image in the background. Once duplication of the base image finishes, instead of merging COW slice back to the original base, we merge the COW slice to the duplication of the base, removing the dependencies between the parent's base image and children's COW slices. We changed existing *qcow* to work as a buffer of disk updates supporting dynamically merging to any duplicated copy of its original base. Thus, the time-consuming disk duplication can be hidden as a background job. An externally synchronous file system has been proposed by Edmund et al. [59] to amortize modifications across a single commit where only external output will trigger file modifications to be committed. Similarly, our COW slice can be regarded as the buffer of modifications, the commit will be triggered when the duplication of base image is done. Figure 3.7 shows the synchronization of disk IOs when a new VM is created from a substrate. Each VM is assigned a COW slice initially, but will have its own independent disk partition in the long run. Step 1 groups multiple modifications before committing the changes to the disk. Step 2 and step 4 represent retrieving data from the base image and the COW slice respectively. Step 3 and Step 5 show that disk changes are synchronized to the COW slice. When a request of creating a new VM is received by a cloud manager, the duplication of the base image file is started as a background job. Other than synchronizing the COW slice to original base image, we synchronize the changes to new base image which is shown in step 6. After merging COW slices to the new base image. VM starts to read and write data

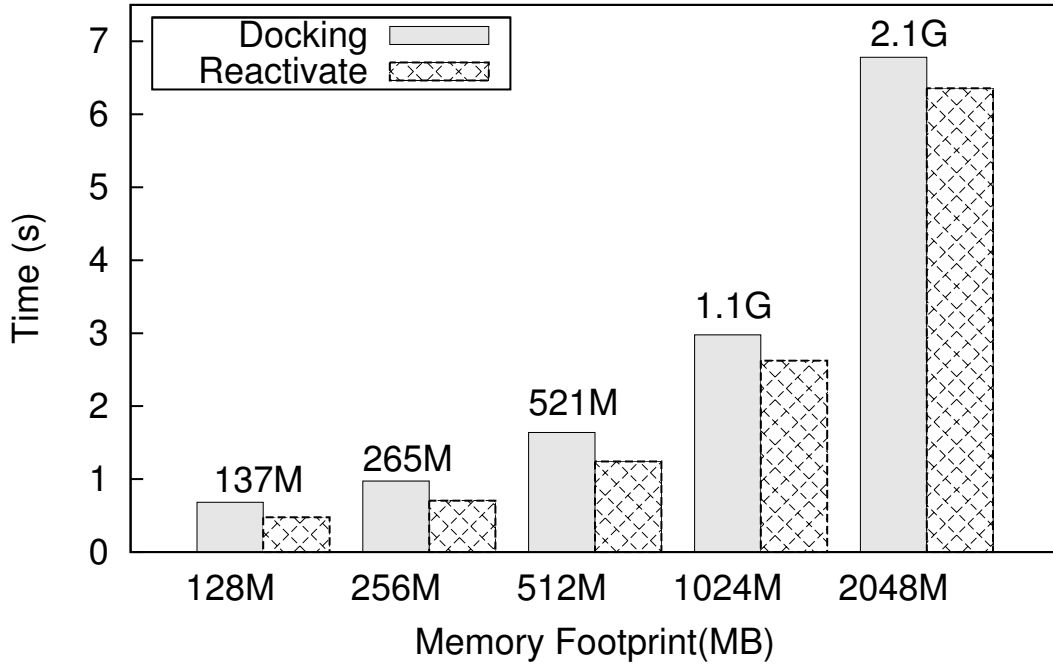


Figure 3.8: Impact of shrinking degree.

directly from and to the new image as shown in step 7 and step 8. After step 8, the VM creation process finishes and the VM works just as the VMs created from a static templates. Note that the VMs created from substrates are online whenever the COW slices are ready (step 1), which gives almost real time responses to users' requests. In practice, the intermediate COW slices turn to be very small after merging, thus can be discarded with minimal cost. The original base image still remains reusable.

Evaluation To understand the impact of shrinking degree on generating VM substrates and reactivating substrates, we shrunk a VM's memory from different sizes. We experimented with various memory sizes from 128M to 2GB and verified the time spent on preparing raw VM substrates and the time reactivating them. All the cached data was synchronized back to disk before docking. As shown in Figure Figure 3.8, the

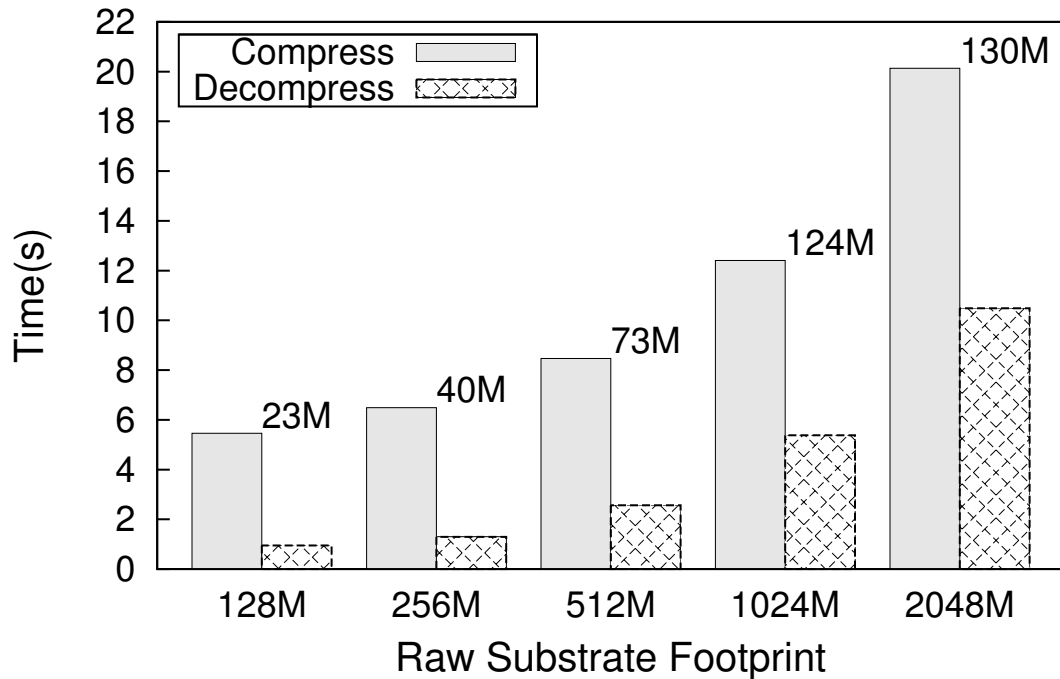


Figure 3.9: Compression cost.

sizes of a raw substrate are slightly larger than the memory footprint. If VM's memory can be shrunk to around 128M, the docking or reactivating can be done within 0.875 seconds. In our test, a VM with some applications like Webserver, MySQL database or program development environment installed could further be compressed, leading to a final VM substrate as small as 16MB.

3.4.2 Substrate Multicast and Compression

In our prototype implementation, we use multicast to dispatch VM substrates in parallel to other physical hosts. Traditional point-to-point communication has the drawback of inefficiency if a substrate needs to be sent to multiple hosts simultaneously. The transferring of VM substrates consumes considerable network bandwidth. In order to make sure that all the VM substrates are only transferred within the data

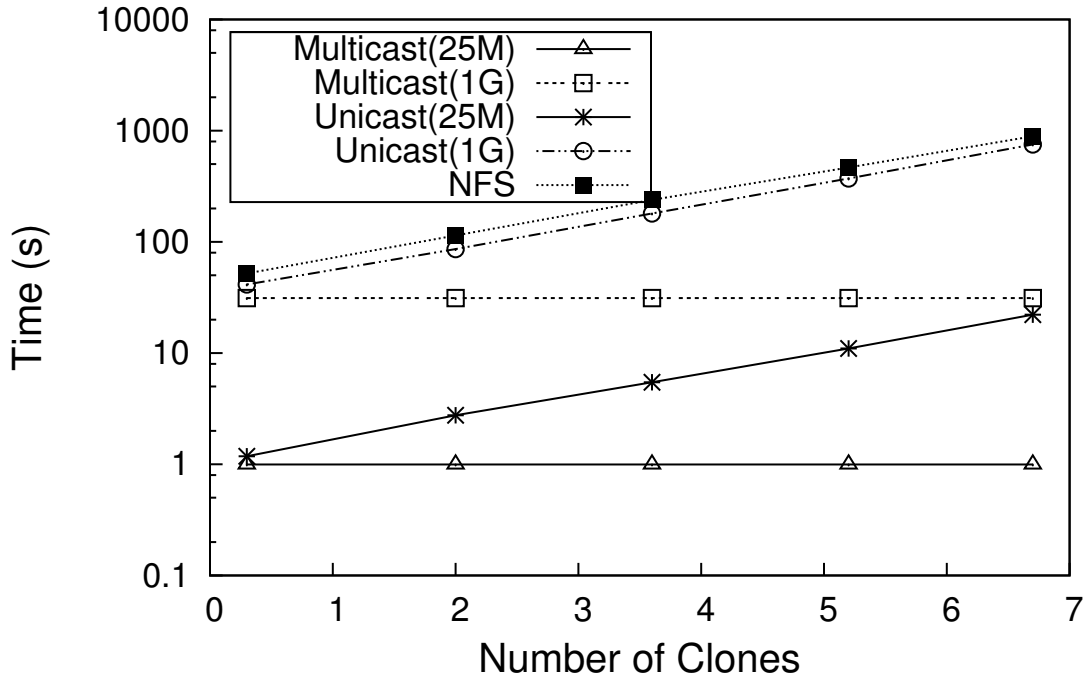


Figure 3.10: Effectiveness of multicast.

center, we set the time-to-live (TTL) value of all multicast packets to be 1. Since the size of VM substrates can be as small as 16MB, the multicast packets can be encapsulated into the payload of TCP packets and can be sent quickly to another node in a LAN environment. Due to the small footprint of the substrates, our current implementation can also be extended to a WAN environment connecting different data centers.

The raw VM substrates are compressed before moved to a pool. The objective of the compression is to make each VM substrate as small as possible. In our prototype implementation, we used the gzip algorithm to compress raw VM substrates. In order to reduce the cost of compression, the compression is done in memory and the resulted compressed substrates are also stored in memory temporarily before they are moved

to the substrate pool. In our experiment, a VM with a development environment installed leads to a size of 16MB after compression. Compression of a substrate is more costly than decompression. Decompression usually takes less than half of the time than compression. The small cost incurred by decompression further speed up the launching process of a VM from the substrate pool.

Evaluation. To evaluate the effectiveness of multicast, we compared the time spent on deploying multiple VMs from the same VM substrate. Figure Figure 3.10 shows the strength of multicast, especially when the number of clones increases. In this experiment, we sent two different substrates with sizes of 25MB and 1GB to different physical hosts in order to create a group of new VMs. As shown in Figure Figure 3.10, multicasting a 25MB substrate to different hosts took less than 1 second while sending the 1GB substrate took around 30 seconds. These are two extreme cases. In the more general case, VM's memory should be able to be shrunk to between 128MB and 1GB, most VMs with barely application environment installed could be shrunk to less than 200MB memory. Thus, the total time on multicast is in the magnitude of several seconds. On the other hand, in the case of unicast without using multicast, the total time of sending the substrates to the others would increase with the number of required clones. Figure Figure 3.10 also plots the time of propagating the VM substrate by duplicating the saved state in a networked file system (NFS).

We also evaluated the cost of compression and decompression on the VM startup latency. We compared the time spent on conversion between raw substrate and final substrate when the size of raw substrate varied from 128MB to 2048MB. As shown in Figure Figure 3.9, compression is more costly than decompression. The final size of

each raw substrate is shown in the figure. For a raw substrate of 256M, which is the size for a typical VM after selective memory dumping, the decompression only took about one second. The startup latency incurred by the decompression algorithm does not significantly affect the users' experiences. Although compression is time consuming especially for large size of raw substrates, the compression is usually done before docking to prepare new VM substrate for future use which does not affect VMs' startup.

3.5 Evaluation

In this section, we examine the overhead and design a set of experiments to verify the effectiveness of VM substrate. We begin by examining the overhead of using a substrate to create new VMs, and then go on to explore one typical usage case of offloading mobile computation to a cloud environment. At the end of this section, we compare the cost of launching a VM with different methods.

The machines used in the experiments consist of a server dedicated to the VM pool and a client machine. All the experiments were conducted in a LAN environment connected by a Gigabit Ethernet switch. The physical hosts for the VM pool is a Dell PowerEdge 1950 server with two quad-core Intel Xeon CPU and 8GB memory. The client machine is a PC with dual CPU cores. We used Xen version 3.4.1 as our virtualization platform. Both dom0 and the guest VMs were running CentOS Linux 5.3 with kernel 2.6.18.

3.5.1 Overhead

We began our evaluation by examining the overhead of VM substrate. We study the latency of preparing a VM or VM cluster on demand. Figure Figure 3.11 draws

the time needed to create different number of VMs through VM substrate pool. In this experiment, we prepared several different VM substrates for each type of applications. Whenever a new VM is needed, in order to minimize the time spent on preparing virtual disks, we created a new VM using a temporary COW slice. The root partition of each VM is 4GB and the partition which is used to store the modification is set to 1GB.

In this experiment, we created different numbers of VMs from the same VM substrate and evaluated the absolute cost. The memory size of a raw substrate in this experiment was shrunk to 118MB, leading to the final compressed substrate of 16MB. This is the smallest size we can achieve with minimal installation of the guest OS and necessary running environment. We intend to answer the following questions in this experiment: (a) What is the optimal speedup VM substrate can achieve? (b) Where is the time spent on VM creation? (c) What is the scalability of the VM substrate approach?

Figure Figure 3.11 shows the time for creating new VMs on demand from the VM substrate pool. The time is broken down into four parts: preparing the disk, multicasting substrate over local network, decompressing VM substrate, and activating VM. From this figure, we can see that the total time of creating a single VM from substrate is as small as 2.5 seconds. This time does not contain the time to generate VM substrates. It assumes that the substrate is always available in the pool. This figure shows VM substrate pool is capable of providing prompt response to latency sensitive VM creation requests. When the number of VMs to be created increases, the total latency of the VM creation does not increase significantly. This is due to the use of multicast, which does not incur proportional overhead when the scale in-

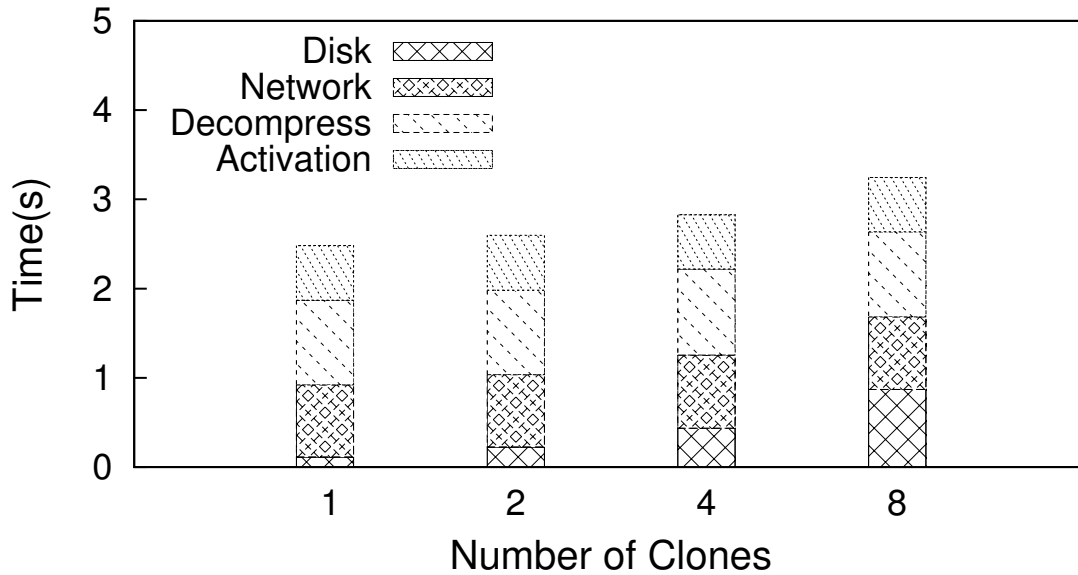


Figure 3.11: Time breakdown of VM creation.

creases. Similarly, the cost of transferring substrates to more than one physical host is almost the same as transferring to a single host. However, the cost of disk creation increases with the number of VMs. Note that the absolute creation time for a single disk is less than a second, given enough storage bandwidth, the disk creation part is not the limiting factor of the scalability of our VM substrate approach.

3.5.2 Performance Comparison

In practice, there are a few different options to start a new VM. These options includes suspend and resume [80], migrating VM from other hosts [25], creating VM from scratch and our VM creation from VM substrate. Among these options, creating VM from scratch involves the whole OS installation process and takes a significant amount of time, which is not considered for comparison.

In this experiment, we created three new VMs containing a web server, a database server and a VM with development environment respectively in the above three dif-

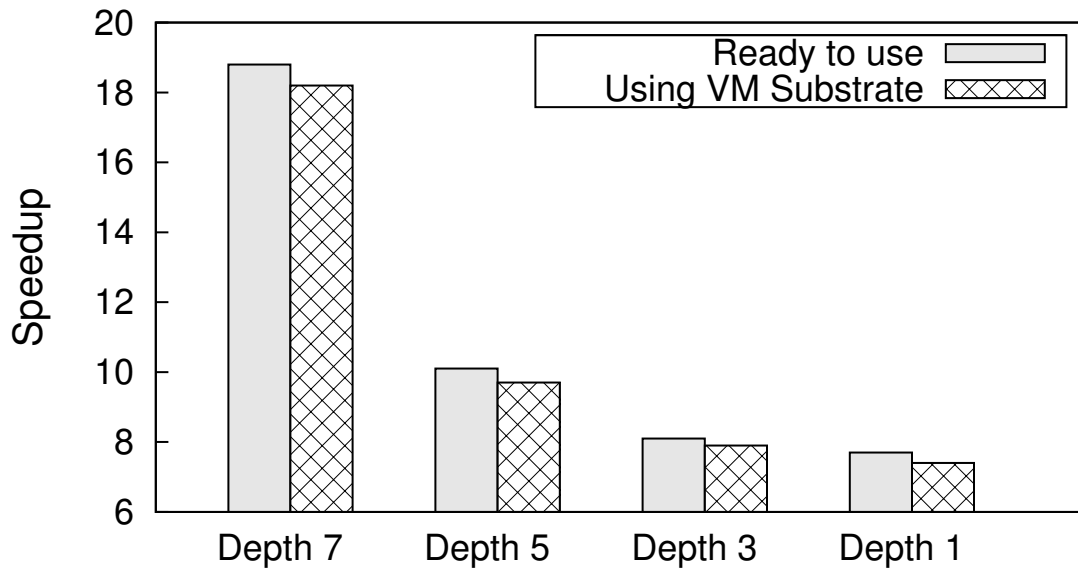


Figure 3.12: Delay time of offloading checker.

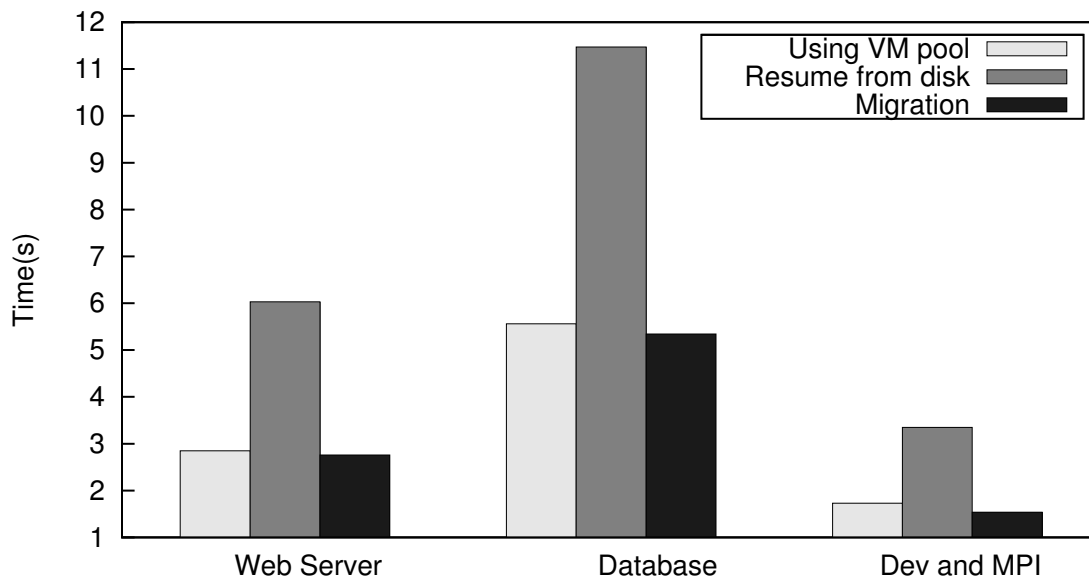


Figure 3.13: Startup time comparison.

ferent ways. Figure Figure 3.13 draws the the startup time of these methods. The startup time is the time between the creation , migration or resume request is received and the time the VM is ready. A VM is considered ready when it is responsive to user's other request like launching a program. Technically, it is when all the virtual CPUs are back online, memory is ballooned back and network interfaces are attached. As shown in Figure Figure 3.13, VM creation from substrate is almost as fast as VM migration. Note that VM migration need to maintain the VM running in its full capacity, which consumes a significant amount of resources limiting the scalability. In contrast, VM substrate maintains a large pool of substrates with mininal footprints. In our testbed with 8GB memory, we were able to host as many as 230 substrates. As expected, the suspend and resume approach incurred considerable startup time because the resume process needs to load a large state file from hard disk.

3.6 Conclusions and Future Work

In this section, we briefly discuss a number of directions that we intend to explore in the future to improve and extend our VM substrate framework. As we have discussed in the previous sections, VM substrate based VM deployment is able to deploy diverse VM within seconds. The idea is preliminary and we plan to further investigate the following areas.

VM streaming. Our current implementation decompress the VM substrate to get the raw substrate and then start new VMs from the raw substrate. Although from Figure Figure 3.9, we can see that decompression takes less time than compression, it is still costly to decompress the substrate when the memory footprint is large. Thus, a mechanism that allows a VM to boot while the decompression is in process will

further reduce the startup latency.

Dynamically linked storage. Because VMs' resources such as vCPU number, memory size and network bandwidth are configurable, it makes the charge of VM resources in pay-as-you-go manner possible. However, storage is not so easily reconfigured as other resources. First, the change of disk size can not take effect without reboot even when LVM is used. Second, running VM's root disk is unable to be altered. Both of these two factors affect the agility of deploying VMs. On the other hand, if each VM can use dynamically linked storage, the actual physical disk partition can be dynamically changed.

Improved memory metering. As discussed in the previous sections, memory footprint is closely related to the final size of VM substrate. The smaller the memory footprint, the smaller the substrate. Our current implementation leverages the `proc` interface under Linux to get the memory utilization. Only the used memory pages need to be dumped in the VM substrate. Identification of unused memory pages or calculation of the memory utilization of a running VM is not trivial. Different from free pages, unused pages refer to those that once touched but not actively being accessed by the system. It can be calculated as the total memory minus the system working set. One possible direction is to integrate more accurate memory metering in VMM level.

In closing, we introduce the primitive of retrofitting VM deployment by using VM substrate and present the design, implementation, and evaluation of a novel approach to manage VMs in agile virtualized environment. Our VM substrate-based VM shrinking and expansion management allows VM creating, reconfiguration in a way that is transparent to users and enables the instantiation of statefull VMs or VM

clusters with sub-seconds latency. Our VM pool architecture is effective in reducing the latency of preparing new VMs and increasing the reusability of VM substrates. It incurs small overhead on the creation of a single or a cluster of VMs. Experiment results on the computation offloading from mobile devices show that the pool of VM substrates is able to provide instantaneous response to user request in an interactive job.

Chapter 4: EFFICIENT SMP VIRTUAL MACHINE SCHEDULING

4.1 Introduction

SMP VMs are ubiquitous in today's scientific computing clusters, modern data centers and cloud computing infrastructures. By consolidating multiple applications on the same underlying physical hardware, cloud service providers benefit from increased hardware resource utilization and the cloud infrastructure management cost. Meanwhile, endusers get the flexibility to pay the cloud services as they scale the VM cluster size base on their workload on-the-fly. Public infrastructure-as-a-service (IaaS) providers like Amazon's EC2 provides extra large instances each with as many as 16 virtual cores [1]. Though modern OSes can be seamlessly running inside a VM with techniques like para or full virtualization [90, 88] and endusers are able to run their applications as if they are running on native OSes, cloud providers are still facing the challenge of consolidating more VMs to reduce the cost and scheduling all these vCPU resource to achieve overall the best performance.

SMP VM blurs the distinction between a virtualized environment with multi-core vCPUs and a physical multi-processor system, imposing a great challenge to vCPU scheduling. Commodity OSes often use spin-locks for exclusive access to shared code or data [83]. Such spin-locks require running processes to frequently acquire and release locks, while assuming only a short period of waiting time. They save the latency cost in circumstances such as interrupt service routines when yielding pCPU for context switch [81] is needed. However, in a virtualized environment, it is hard to

keep the assumption when a vCPU is preempted while still holding a spin-lock and at the same time another sibling vCPU is still waiting for the spin-lock. Thus the sibling vCPU has to wait until the preempted vCPU to be rescheduled and releases the lock. Such switch between sibling vCPUs wastes large amounts of CPU cycles (usually in the order of a few milliseconds) and causes severe performance degradation, particularly when the waiting vCPU has been scheduled multiple times before the release of the lock. Such phenomenon is unique in multicore VM environments and often referred to as lock holder preemption (LHP) [83].

To solve the LHP issue, one solution is to detect the lock holder and avoid preemption. Lock holder could be detected by instrumenting guest OS's spin-lock primitives in para-virtualization [83] or by leveraging hardware techniques [94]. Once lock holder is detected, hypervisor's scheduler either avoids preempting lock holder or delays the lock waiter for the purpose of minimizing the synchronization latency [28]. This lock holder detection and avoidance technique is beneficial to the cases where spin-lock is infrequently involved. However, it still requires either the change of the guest OS itself or the support from low-level hardware. The lock holder detection itself also cause VM's response latency.

The VM LHP issue could be addressed by co-scheduling [65, 89]. In SMP VM co-scheduling, the sibling vCPUs are co-scheduled on pCPUs simultaneously. This gives the guest OS an illusion of running on a dedicated server with the same number of processors. Co-scheduling improves performance by facilitating prompt communication and reducing synchronization delay between sibling vCPUs. For example, if one vCPU A is spinning on a lock waiting for another vCPU B to release the lock, co-scheduling A and B allows the spinning vCPU A to proceed as soon as B releases

the lock without waiting for the preempted vCPU to get rescheduled. A few recent work applied co-scheduling to SMP VMs running concurrent tasks [16, 84]. Such proactive solutions are favorable to applications heavily relying on spin-lock and their performance gain outweighs the overhead of co-scheduling. However, co-scheduling often comes with side effects, such as CPU utilization fragmentation, execution delay and priority inversion [81]. These potential effects limit the massive use of SMP VM co-scheduling.

In this work, we propose a new approach called SBCO for performance optimization in virtualized SMP environments. SBCO inherits the advantages of traditional co-scheduling such as minimizing synchronization latency and accelerating communication between vCPUs without the side effects of scheduling fragmentation and priority inversion. SBCO does not force simultaneously co-scheduling all the sibling vCPUs. Instead, it re-adjusts the sibling vCPUs positions in their respective run queues and facilitates sibling vCPUs to be scheduled at the same time window. Specifically, SBCO first dynamically adjusts the affinity between vCPUs and pCPUs to prevent sibling vCPUs from being assigned to the same run queue. By distributing sibling vCPUs evenly to different run queues, it significantly reduces the chance of stacking sibling vCPUs. Then through minimizing the scheduling distance of sibling vCPUs defined in Section 4.3.3, our approach reduces the maximal scheduling distance and further reduces synchronization latency. We have implemented SBCO in KVM, and performed extensive evaluations with both micro-benchmarks and real-world workloads. The experimental results show that SBCO can significantly reduce the number of vCPU context switches and achieves an overall performance improvement by more than 10%.

4.2 Background

An SMP VM is able to leverage the multicore processors to execute multiple independent workloads simultaneously. Thus, SMP VMs are widely used by cloud service providers. The vCPUs of a SMP VM are usually attached to processes or threads on a physical host and they execute codes by direct code execution or instruction emulation [88]. vCPUs are scheduled as processes or threads on a host OS. Therefore, there are two layers of scheduling, where the hypervisor schedules vCPU threads on pCPUs and a guest OS schedules tasks on vCPUs. vCPUs are usually dynamically mapped to pCPUs and the co-scheduling of sibling vCPUs is essentially co-scheduling of vCPU threads on pCPUs. Such dynamic mapping improves hardware utilization by balancing workloads between pCPUs. However, it also causes sibling vCPU stacking issue, elaborated in Section 4.3.

In a parallel program, a lock primitive is needed to provide synchronization and guarantee atomic and consistent state changes in a multiprocessor system. There are typically two types of lock primitives: *semaphore/mutex* and spin-lock [65]. The former lock primitive blocks the running process until the required resources or locks become available. The scheduler swaps out the running process(unless specifically stated, we use the term process, thread and task interchangeably) and immediately schedules the next runnable process so as to avoid wasting CPU cycles. It needs process context switches to wake up the sleeping process, thus degrading system performance. In contrast, spin-lock allows the thread waiting for the required resource to keep occupying the processor and repeatedly check the lock status. It works efficiently when synchronization only takes a small amount of time(usually dozens of

Table 4.1: Statistics of spin-lock Usage

Lock	Metric	SPECjbb	SupperPI	KernBench
spin_lock	M1	0.160 μ s	0.191 μ s	0.650 μ s
	M2	7,256,276	1,821,548	371,309,982
spin_lock_irq	M1	0.115 μ s	0.150 μ s	0.164 μ s
	M2	5,312,485	1,305,679	146,834,593

M1: avg lock holding time **M2:** num of total call

or hundreds of microseconds). Because the lock efficiency directly affects system performance and capability, spin-lock is widely used in modern OSes.

Spin-lock poses challenges in SMP VM scheduling. It works effectively in the cases that the lock holder only holds the lock for a very short period of time and the target resources become available soon. This is satisfied in physical environments where OS itself has control over the resources and the way of scheduling via determining whether or not to preempt out a process. However, in a virtualized environment, it is the virtual machine monitor(VMM) that retains ultimate control of the resources and vCPUs scheduling usually based on time slices. Thus the current spin-lock design may not be effecient. For instance, if a vCPU is trying to acquire a spin-lock, it has to wait until the preempted vCPU is scheduled back and release the lock. Such phenomena, referred to as LHP issue, significantly increases the lock holding time and may even waste a vCPU's time slice, especially in CPU over-committed cases. The high vCPU contention from a preempted lock leads to significant waste of CPU cycles [83].

To study the cost of spin-lock in a virtualized environment, we ran three different workloads, SPECjbb, SupperPI and KernBench(see Section 4.6.1 for workload specifications) in a VM and instrumented host machine's Linux kernel(version 2.6.34.4) with a kernel tracing tool Ftrace [20] to track lock usage statistics. Based on our experiment, *raw_spin_lock* and *raw_spin_lock_irq* are the two lock functions contributing

to the majority of the total busy-waiting lock holding time. We sampled the execution time of these two lock functions in a three second period, repeated the experiment for five times and summarized the results as follow. As shown in Table Table 4.1, metrics 1(M1) represents the lock holding time and metrics 2(M2) is total number of call of these lock functions. We observe *spin_lock* is heavily used in all workloads, and the average lock holding time ranges from 0.160 μ s to 0.650 μ s. *spin_lock* consumes 8.05% of the whole execution time in KBench case. Different from *raw_spin_lock*, *raw_spin_lock_irq* requires disabling interrupt before holding the lock. Though the operation has less lock holding time than the former, based on the numbers showing in the table, it still leads to wasting almost 1% of the total execution time on locks. Another observation is that the CPU-intensive workload SupperPI involves less locks and less average lock holding time compared with the mixed kernel compile workload. These results suggest that some scheme to reduce spin-lock cost is deemed necessary. Our tentative solution is introduced in Section 4.4.

4.3 Challenges

In this section, we first elaborate a few challenges of SMP VM scheduling caused by virtualization abstraction layer. Then, we introduce a new concept called scheduling distance, analyze its effect on synchronization latency and present a few motivation examples for our new approach.

4.3.1 Dynamic vCPU Affinity

A VM's vCPU affinity configuration is one of factors complicating SMP VM scheduling. In a typical physical environment, there are generally two types of CPU affinity: strict affinity and soft affinity. Strict affinity tends to keep a process on the

same CPU as long as possible by exclusively limiting the options of CPUs a process can run on. Enforcing hard affinity is crucial to cache performance, especially in performance-critical situations like a large database or a multithread java server, because data can be maintained in only one processor's cache at a time. Otherwise, if a process is executed among multiple processors, whenever a processor updates its local cache, the rest processors with the same copy of the data have to invalidate that cache or update depending on the cache coherence protocol. Such cache synchronization among processors become costly when a process keeps bouncing between processors and cause frequent cache invalidations or updates. In contrast, soft affinity tends to balance the load between CPUs and migrate a process to a less busy CPU. It avoids imbalanced scheduling and greatly increases CPU utilization. In a dynamic multithreaded environment, it is impractical to manually or programmatically bind a thread to a CPU without jeopardizing overall system utilization efficiency.

VMM controls the scheduling of vCPUs to balance vCPUs among pCPUs. It is the guest OS that decides the scheduling of the processes inside a VM with the objective of balancing them between vCPUs. VMM usually does not distinguish vCPUs from different VMs and the default scheduler often employs a global load balance policy by scheduling processes to less busy pCPUs. Such policy keeps the balance of utilization between different pCPUs because of no limit of default affinity. The randomness of affinity is likely to have one or more sibling vCPUs scheduled in the same run queue. This is usually referred to as vCPU stacking issue [81]. In addition, if a VM process changes its processor, it only changes its vCPU and does not guarantee the change of low-level pCPU. Many of today's hypervisor like KVM [41], Xen [96] or VMware ESX [6] allows statically configuring the mapping between vCPUs on

pCPUs. However, due to the dynamics inside a guest VM, such affinity configuration on the host is commonly used for CPU resource reservation and VM isolation, rather than for the purpose of applications' performance or resource utilization efficiency.

Though stacking of sibling vCPUs is a probability type of issue, it greatly increases the lock synchronization latency in a virtualized environment. If stacked sibling vCPUs are competing for the same resource using spin-lock, the sequential vCPU execution would waste significant amounts of CPU cycles [81]. The probability of stacking sibling vCPUs in CPU intensive workload case was studied in [81]. Their experimental results reveal that the chance of more than one sibling vCPU in the same run queue reaches as high as 45% when three CPU intensive VMs were consolidated on the same server [81]. We further conducted complementary experiments to examine the vCPU stacking issue with IO intensive and CPU-I/O mixed workloads such as SPECjbb and Kernbench. The details of these workloads are introduced in Section 4.6.1. We implemented an independent kernel thread to periodically examine each pCPU run queue with an interval of one second. Then, we ran a kernel compile benchmark and SPECjbb in a number of VMs and counted the number of samples when more than one vCPU sibling exists in the same run queue. Table 4.2 shows the accumulated probability of stacking vCPUs can be higher than 20% for both workloads. In the case with three VMs, the probability can go beyond 42% with the Kernbench workload. Stacking sibling vCPUs can greatly increase the chance of having LHP issue. Such high stacking ratio can even break an illusion of synchronous progress of vCPUs, which is expected from a guest OS [88]. Without this illusion, synchronization latency significantly degrades applications' performance.

Table 4.2: Probability of stacking sibling vCPUs

Apps	2 VMs	3VMs
SPECjbb	20.25%	31.63%
Kernbench	33.19%	42.84%

4.3.2 Costly vCPU Context Switch

vCPU context switch is another challenge of SMP VM scheduling. In parallel system, multiple processes may share a pCPU and frequently involve context switches. In each context switch, a pCPU needs to save and restore its state; the TLB entries need to be reloaded; and processor pipeline must be flushed; the OS kernel scheduler must execute [50]. Besides these unavoidable cost for each context switch, due to pollution of processors' cache, virtual memory maps need to be re-synced, which results in some indirect penalty of performance when cache miss happens. This indirect cost varies for different workloads with different memory access patterns. Due to the fact that each vCPU is associated with additional data structures to maintain information like the status of virtual registers, scheduling a vCPU thread causes uncertainties to the indirect cost. Even if a vCPU thread is scheduled back to the original pCPU, inside the guest VM, the vCPU may be serving a different task, which results in invalidation of pCPU cache. To evaluate the additional cost of context switching in a virtualized environment, we ran a context switch micro benchmark [7] in a KVM guest VM and examine the average cost of context switch and system call. As shown in Figure Figure 4.2, on average, vCPU context switch costs 2.5 to 3 times more in a virtualized environment compared with the a physical environment. Therefore, effective vCPU context switch is one of the design goals of our SBCO approach.

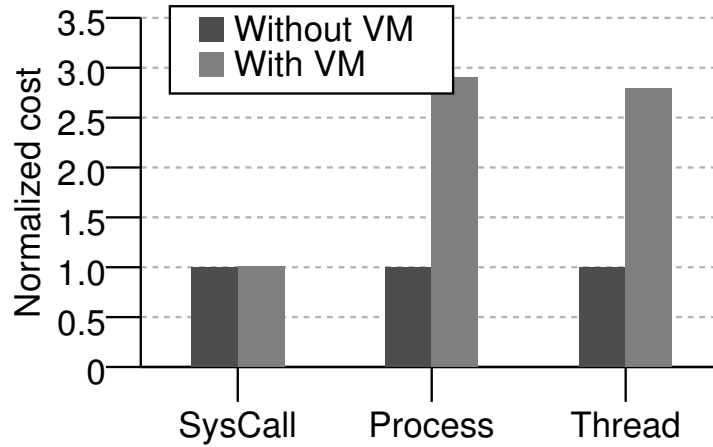


Figure 4.1: Cost of vCPU context switch w/ and w/o virtualization.

4.3.3 The Effect of Scheduling Distance

A commodity scheduler commonly splits up pCPU time between runnable processes in a fine-grained way in the order of nanosecond accurate time slices. Recall that sibling vCPUs could be stacked in the same run queue of a pCPU. Let P_{run} denote the current total number of processes in a run queue and T_w be a process's dynamic priority, also referred as the weight. T_{wi} is the weight of process i in a run queue. Let S_{min} be *sched_min_granularity*, the minimum time a task will be allowed to run on CPU before being forcibly preempted out. Let $S_{latency}$ be *sched_latency*, the default scheduling period in which all run queue processes are scheduled once. S_{min} and $S_{latency}$ are configurable parameters in the default scheduler. T_{slice} is the time slice of a process with the weight of T_w . Assuming all the processes in a pCPU run queue have the same weight, then each process also has the same time slice T_{slice} . The actual scheduling period T_p , which is the total time all run queue processes are scheduled once, is calculated in the following formula. T_p is also the maximum time

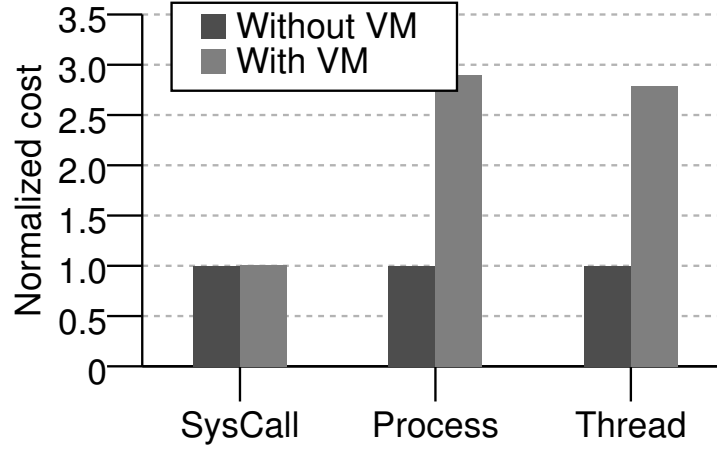


Figure 4.2: Cost of vCPU context switch w/ and w/o virtualization.

one process has to wait until all other process to yield pCPU. For instance, if a VM has two vCPUs A and B stacked in the same pCPU run queue. Assuming A is at the front of the run queue and B is at the tail of the run queue, in the worst CPU intensive workload case, B has to wait for T_p after A gets the chance to be scheduled.

$$P_{total} = S_{latency} / S_{min};$$

$$T_p = \begin{cases} S_{latency} & P_{run} \leq P_{total}; \\ P_{run} * S_{min} & Otherwise. \end{cases}$$

$$T_{slice} = T_p * (T_w / \sum_1^n T_{wi}), i \in [1, P_{run}].$$

The scheduling time can be viewed as an axis with the time to schedule a vCPU as the origin, and time slots when vCPUs will be scheduled as scheduling ordinates. We define a VM's schedule distance as the maximal difference of sibling vCPUs' scheduling ordinate. The latter is also the relative position in pCPU run queues. As

the illustration example shown in Figure Figure 4.3, VM1 and VM2 are running on a physical machine with two pCPUs. At T_0 , both of VM1's vCPUs are in a pCPU's run queue and are ready to run. Moreover, VM1's $vCPU_0$ is the next candidate to be run on $pCPU_0$ and will be scheduled immediately when $pCPU_0$ becomes available. However, VM1's $vCPU_1$ is currently at the bottom of $pCPU_1$'s run queue and does not start to run until T_n . Let $Pos(t)$ denote process t 's position in its run queue and $L(n)$ as the length of $pCPU_n$'s run queue. We define *delta*, also denoted as $D(VM1)$ in Figure Figure 4.3, as the maximum difference of scheduling distance disparity between sibling vCPUs as follows:

$$Pos(vCPU_0) = 1;$$

$$Pos(vCPU_1) = L(pCPU_1);$$

$$delta = |Pos(vCPU_0) - Pos(vCPU_1)|.$$

Though $vCPU_1$ is runnable in this case, if $vCPU_0$ is waiting for $vCPU_1$ to release the lock, then $vCPU_0$ has to wait for $pCPU_1$ to reschedule task $vCPU_1$. At the same time, $vCPU_1$ has to wait for processes before it to acquire pCPU, execute for T_{slice} and then yield pCPU for reasons like waiting for IO or using up its own time slice. The waiting time could be as long as T_p in the worst case, which is often the order of tens of milliseconds. Depending on the length of the run queue and how long a task typically run before getting switched out again, it can considerably degrade performance, especially in the dense consolidation of CPU intensive workloads, in which the average run queue size is usually large. A VM's scheduling distance can greatly increase synchronization latency even if the sibling vCPUs are dispatched to different run queues without stacking vCPU.

$$D_{threshold} = \alpha * Q_{size}, \alpha \in (0, 1).$$

We investigated VM's scheduling distance by running a few VMs with the average run queue size as six and ten respectively. We implemented an independent kernel thread to periodically check each pCPU's run queue and simply count the cases that a vCPU is ready to be scheduled but with one or more sibling vCPUs having $D_{threshold}$ distance in the respective run queue. $D_{threshold}$ is equal to α times of a pCPU run queue length, denoted as Q_{size} in the above formula. Note that a big α value leads to large $D_{threshold}$, and less probability of exceeding the threshold. In contrast, a small α may cause frequent adjustment for balancing sibling vCPUs. We ran Kernbench and NPB benchmarks (refer to Section 4.6.1 for detailed benchmark introduction) and computed the probabilities for sibling vCPUs to exceed different $D_{threshold}$. As shown in Table 4.3, the probability of exceeding $D_{threshold}$ with α as $2/3$ is between 40% and 50% for both workloads when the average run queue size is six and ten respectively. This result suggests that α as $2/3$ is a good tradeoff. To alleviate the synchronization latency problem from the LHP issue, we propose SBCO which leverages the scheduling distance information to make scheduling decisions.

4.4 Self-Boosted Co-Scheduling

Unlike conventional VM co-scheduling in which sibling vCPUs are scheduled at precisely the same time, SBCO shortens the scheduling distance between sibling vCPUs. SBCO preserves the flexibility of dynamic vCPU affinity and reduces the costly vCPU context switch. In this section, we elaborate the design details, and discuss a few optimization techniques of the design.

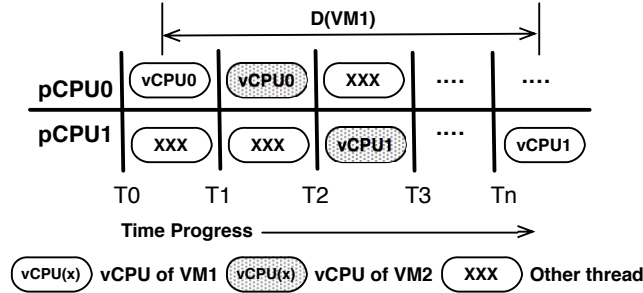


Figure 4.3: Scheduling distance sibling vCPUs.

Table 4.3: Probability of scheduling distance exceeds the threshold

RQ size	Probability of exceeding $D_{threshold}$	
	Kernbech	Parsec
6	42.36%	40.75%
10	48.21%	45.13%

4.4.1 Overview

As studied in Section 4.3, excessive busy-waiting `spin-lock` holder preemption and vCPU stacking lead to parallel applications' performance slowdown. Classic co-scheduling incurs heavy context switches due to forcibly preempting pCPUs to schedule vCPUs from the same VM. Our SBCO is designed to reduce costly vCPU context switching and shorten the synchronization latency caused by `spin-lock` holder. It maintains a balance between a fast vCPU with a large scheduling ordinate and a slow sibling vCPU with a small scheduling ordinate. Therefore, we try to answer the following questions when designing SBCO. 1) How to avoid stacking sibling vCPUs? 2) How to flexibly balance a fast vCPU and a slow sibling vCPU? 3) How to avoid forcibly preemption and reduce vCPU context switches? 4) How to control SBCO's overhead while keeping its efficiency? In the following section, we first further clarify a few new concepts mentioned this work with some illustration examples, then we discuss the details of the design of extended RB tree, SBCO algorithm as well as some

RBINDEX	PID	VRUMTIME	PARENT_VRUN	COMMAND	ORDER
<0>	3173	[5556.399127]	--> [5556.792013]	qemu-kvm	((0))
<0>	3161	[5556.620530]	--> [5556.399127]	cpuhog	((1))
<2>	3177	[5556.792013]	--> [0]	cpuhog	((2))
<0>	3164	[5557.007206]	--> [5557.192115]	qemu-kvm	((3))
<1>	3159	[5557.192115]	--> [5556.792013]	cpuhog	((4))
<0>	3181	[5557.198289]	--> [5559.010627]	qemu-kvm	((5))
<1>	3168	[5559.010627]	--> [5557.192115]	cpuhog	((6))
<0>	3187	[5560.140783]	--> [5559.010627]	qemu-kvm	((7))

Figure 4.4: CFS run queue snapshot.

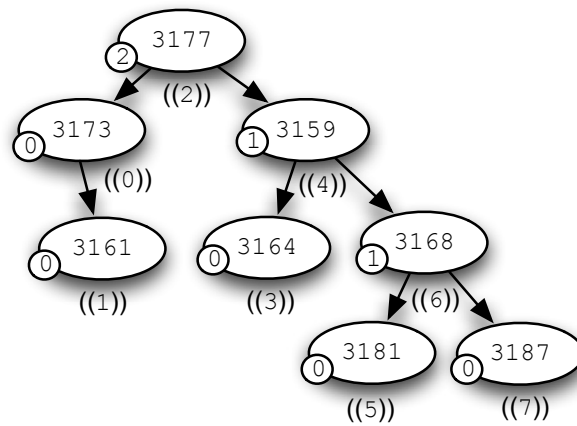


Figure 4.5: Extended red black tree with index.

performance optimization considerations.

4.4.2 Extending Red Black Tree

In existing Completely Fair Scheduler(CFS) scheduler, each pCPU has an independent run queue. All the processes in a pCPU run queue are managed with a self-balanced binary search tree called read black tree(RB tree) [2]. The process with the smallest `vruntime` (virtual runtime in nanoseconds), which corresponds to the left most node in the RB tree, is chosen by the scheduler as the next candidate to run. The default RB tree is constructed starting from the arrival of the first process in a run queue. New processes with smaller `vruntime` will be placed before the left most

child and the tree will rotate itself to keep balanced. When a process finishes running, its `vruntime` with the total execution time weighted by its priority is updated. Once a process leaves its run queue, the associated node will be removed from the RB tree of that run queue and the RB tree will also rotate to keep balanced.

The default RB tree does not maintain processes' relative positions in a run queue. Instead, it simply sorts processes according to their `vruntime`. Therefore, it involves RB tree traversal in order to calculate how many processes in a run queue are ahead of a process, which is contradicting to the simple but efficient design philosophy of scheduler design. We solve this dilemma by extending the default RB tree data structure by adding `RB index` when constructing a RB tree. The `RB index` of a RB tree node is defined as the total number of nodes in the left child sub-tree of this node. We summarize all the terms we mentioned as following:

RB index: The total number of nodes in the left child sub-tree of a node in a RB tree. It is updated with RB tree balance rotation when there is a node added into or removed from a run queue.

Scheduling ordinate: A vCPU process's position in its run queue. It reflects the maximum number of processes ahead before a process gets scheduled.

Scheduling distance: The difference between of the fastest vCPU's `scheduling ordinate` and the slowest vCPU's `scheduling ordinate` in a VM. Figure Figure 4.4 gives a snapshot of a pCPU run queue. To demonstrate the RB tree of a run queue, we run a four vCPUs KVM VM with CPU intensive workloads to keep all vCPUs busy. Meanwhile, we ran a four threads application(`cpuhog`) on the host machine to represent non-vCPU threads in the pCPU run queue. In reality, these non-vCPU threads could be kernel threads or any applications running together with a VM hy-

Algorithm 1 SBCO Main Algorithm

```

1: procedure SBCO(VOID)
2:    $a \leftarrow$  left most task in RB tree
3:    $n \leftarrow a$ 
4:   if Task  $a$  is a vCPU task then
5:      $orda \leftarrow$  SCHED_ORDINATE( $a$ )
6:     for all Task  $t$ 's sibling task  $b$  do
7:       if Task  $b$ 's dirty flag is set then
8:         continue
9:       end if
10:       $ordb \leftarrow$  SCHED_ORDINATE( $b$ )
11:       $delta \leftarrow$  abs( $orda - ordb$ )
12:      if  $delta \geq D_{threshold}$  then
13:         $n \leftarrow$  Task  $a$ 's successor in the run queue
14:        SCHED_BLANCE( $t, b$ )
15:        break
16:      end if
17:    end for
18:  end if
19:  return  $n$ 
20: end procedure

```

pervisor. All the processes's information, including process name(command column), process id(pid column), virtual runtime(vruntime column) and parent process's virtual runtime, are listed in the figure. All the vCPUs in a KVM VM have the same process name(qemu-kvm) . Based on the virtual runtime relationship between child process and parent process in the figure, a RB tree is constructed in Figure Figure 4.5. The **RB index** column in Figure Figure 4.4 and each number in the small circles in Figure Figure 4.5 represents the total number of nodes in the left child sub-tree of a process. Based on **RB index**, **scheduling ordinate** of a task in its run queue is calculated by Algorithm 4.4.3 and printed out in the right most column in Figure Figure 4.4 and in the brackets under each tree node in Figure Figure 4.5.

In conclusion, by introducing **RB index**, the scheduling ordinate of a process is calculated with $O(\log(n))$ complexity. The scheduler can spend minimal amount of time on choosing next vCPU process to run while taking sibling vCPUs' scheduling ordinate into consideration. The details of complexity analysis and performance considerations are provided in the following subsections.

4.4.3 SBCO Algorithm

Algorithm 1 shows the pseudo code of SBCO. For each scheduling period, SBCO first chooses a task with the smallest **vruntime** in a run queue as the default candidate to run (line 2). If the current candidate is a vCPU process, which is implied by the process's name, SBCO then calculates this vCPU's scheduling ordinate (line 5), iterates other sibling vCPUs to identify if there is any runnable sibling vCPU with large scheduling distance and decide if it is necessary to balance the fast and slow sibling vCPUs. If the scheduling distance between two sibling vCPUs, calculated in line 11, exceeds the $D_{threshold}$, which indicates the candidate vCPU runs too fast,

then there is a need to enforce adjustment to delay the fast vCPU and speed up the slow one(line 13). As a result, the previously selected candidate vCPU, the default left most node, is no longer the next task to run. Instead, the scheduler chooses the candidate's next successor process in the RB tree to run.

Note that each vCPU process is guaranteed one time to be scheduled in a scheduling period T_p , defined in Section 4.3.3, we design two approaches to eliminate repetitive adjustment on one vCPU and ensure each vCPU being scheduled once in T_p respectively. First, we mark those sibling vCPUs that have been already adjusted as dirty. This dirty tag aims to prevent a vCPU thread from repeatedly yielding its pCPU. The adjustment is realized as follows: the vCPU with the smallest scheduling ordinate lends certain amount of `vruntime` to the sibling vCPU with largest scheduling ordinate, causing both move towards the center of their respective run queues. When the scheduler decides a task to run, it first checks a vCPU's dirty tag and it will not re-balance with the sibling vCPU marked as dirty. Second, as shown in function `SCHED_DISPATCH`, each VM's sibling vCPUs are dispatched to different pCPU run queues, preventing them from the stacking issue. But we let the default load balancing to take over the control of the mapping between a pCPUs and a vCPUs. This still maintains the physical resources utilization efficiency.

4.4.4 Performance Considerations

We have following design considerations to minimize the overhead of SBCO: 1) maintain the RB index for each vCPU. 2) set dirty tag for balanced vCPUs. 3) maintain a debt list for the adjustment between sibling vCPUs. In the following section, we analyze these design considerations and their tradeoff.

RB index. Instead of directly keep each vCPU process's scheduling ordination in pCPU run queues, SBCO seamlessly inserts RB index information into the existing tree structure. Due to the fact that each vCPU process is swapped in to or be swapped out from a run queue frequently during the execution, the scheduling ordinate of a vCPU is constantly updated with the change of its position in its resident run queue. There are two advantages to introduce RB index. First, RB index can be used to efficiently calculate the scheduling ordinate. As show in function *SCHED_ORDINATE* in Algorithm 2, the calculation of a vCPU's scheduling ordinate only involves RB tree traversal from root to vCPU's corresponding node and the complexity is bounded to $O(\log(n))$. Second, a vCPU's RB index is updated dynamically with the RB tree rotation. This update only involves the change of the nodes on the path from root to the node. The additional cost on operating RB index is only limited to assigning value to the *rb_index* in the data structure without any extra lock.

Debt list. It is very costly for the scheduler to hold the locks of two run queues while changing one of them, such as migrating processes. In order to avoid locking two run queues at the same time when conducting the adjustment, we maintain an independent *debt* list for each pCPU run queue. Therefore, changing a *debt* list does not require to acquire that run queue's lock. As shown in function *SCHED_BLANCE* in Algorithm 2, when balancing task T_a and T_b , T_a 's *vruntime* is adjusted and its location in its tree is updated immediately. However, T_b 's *vruntime* and location are recorded in the associated pCPU *debt* list temporarily. The change is delayed to the time when the scheduler needs to choose a process from T_b 's resident pCPU run queue. As a result, the actual balancing is conducted in two different times, which avoids locking two run queue simultaneously. In addition, since the scheduler

Algorithm 2 SBCO balance and RB ordinate algorithm

```

1: procedure SCHED_DISPATCH( $T_a$ )
2:    $cpus \leftarrow$  all pCPUs
3:   if  $T_a$  is a vCPU then
4:     for all Task  $t$ 's sibling task  $b$  do
5:        $cpu\_occupied \leftarrow b$ 's pCPU
6:        $cpus \leftarrow cpus - cpu\_occupied$ 
7:     end for
8:   end if
9:   return  $cpus$ 
10: end procedure

11: procedure SCHED_BLANCE( $T_a, T_b$ )
12:   if  $T_a$  is clean then
13:     Adjust  $T_a$ 's vruntime
14:     Reposition  $T_a$  in its RB tree
15:     Update the debit list of  $T_b$ 's run queue
16:   end if
17: end procedure

18: procedure SCHED_ORDINATE( $T_a$ )
19:    $parent \leftarrow T_a$ 's parent task in rb tree
20:    $n \leftarrow RBindex_{T_a}$ 
21:   while  $parent \neq null$  do
22:     if  $T_a$  is parent's right child then
23:        $n \leftarrow n + RBindex_{parent} + 1$ 
24:     end if
25:      $T_a \leftarrow parent$ 
26:      $parent \leftarrow T_a$ 's parent
27:   end while
28:   return  $n$ 
29: end procedure

```

has to check its *debt* list and apply the changes to T_b before it chooses a process candidate to run, *SBCO* incurs the additional marginal cost, mainly on updating data structure. The default scheduler does not distinguish a vCPU process from other normal tasks, *SBCO* always checks a task's name when making scheduling decisions, and only balances *qemu-kvm* processes, which are KVM vCPUs. Other non-vCPU processes are ignored for balancing.

Dirty tag. To prevent repeatedly adjusting the same vCPU in the same balance round, which may cause starvation, we mark the changes when a vCPU is adjusted but the changes has not been applied yet, either in the case of being given *vruntime* by or lending *vruntime* to other siblings. For instance, as shown in Algorithm 1 line 7, if a sibling vCPU has been adjusted before, *SBCO* passes that sibling vCPU and continues to check if there is any other available sibling for balancing. After the change recorded in a *debt* list is applied to a vCPU, this vCPU's dirty flag is cleared and the vCPU becomes available again for future balancing. The detailed cost analysis is provided in the evaluation section.

4.5 Implementation

We implemented the prototype of *SBCO* algorithm in KVM with Linux kernel 2.6.34.4. KVM is a user friendly virtualization solution seamlessly integrated into Linux kernel. In KVM, there are two kinds of important threads which are QEMU threads and vCPU threads. The QEMU threads share the responsibility for the actual disk I/O by emulating the hardware devices. The vCPU threads execute the real code. KVM relies on existing Linux scheduler for the scheduling of vCPUs and each vCPU is treated as a normal task in host OS.

Our SBCO algorithm is implemented based on CFS scheduler. We extended the default RB tree to carry RB index and implemented associated APIs to calculate scheduling ordinates. We added new *rb_index* and *rb_dirty* to each node of the RB tree. The *rb_index* keeps the number of nodes on the left side of a node in the RB tree. It is updated during self rotation of a RB tree when a new task is enqueued or an existing task is dequeued. In addition, the *rb_dirty* records if a process is needed to be adjusted. To avoid repeatedly yielding the same process in one balance round, when *rb_dirty* is set, the process is ignored for balancing with its sibling.

Note that a vCPU's run queue is changed in three cases: 1) a vCPU process is created and then inserted to a run queue. 2) a vCPU process wakes up from sleep and needs to enter a run queue. 3) a vCPU is migrated between two pCPUs. We instrumented the scheduler to avoid stacking sibling vCPUs in all these cases. We first modified CFS scheduler to dynamically set a task's *cpus_allowed* field which is a set of pCPUs that a task can run on. It is set before choosing a run queue for a vCPU. This medication solves the first two cases. Then, we changed the scheduler's default load balance function *can_migration* by limiting the options of migration destination pCPUs. Therefore, no sibling vCPUs co-exist in the same run queue even after load balance. In our evaluation, we limited the the number of vCPUs of a VM to be less than or equal to the total number of pCPUs. However, in the implementation, if the number of vCPUs is more than pCPU number, we allowed the rest of vCPUs to randomly select their pCPUs.

For comparison, we also implemented the idea of balanced scheduling and two conventional co-scheduling approaches [81]. The balanced scheduling simply puts sibling vCPUs to different pCPU run queues by adjusting *cpus_allowed* field of their

process structure. In addition, we developed two more co-scheduling approaches. First, when $vCPU_0$ of a VM is scheduled, the rest of sibling $vCPUs$ are forcibly scheduled on other $pCPUs$ concurrently. Second, let $pCPU_0$ decide to co-schedule all the $vCPUs$ of a VM depending on which $vCPU$ the first $pCPU$ will run. We refer these two co-scheduling approaches as PROCCO and CPUCO respectively in our evaluation. In both cases, an inter-processor interrupt (IPI) request is sent to the related $pCPU$ to force context switch and pick a sibling $vCPU$ instead of the default lowest `vruntime` task to run. Given all sibling $vCPUs$ are dispatched to different run queues, we assume that each $pCPU$ run queue size is close to each other. In our prototype, we define the scheduling distance threshold to be $2/3$ times of the size of each run queue, as suggested in Section 4.3.3.

4.6 Evaluation

In this section, we study the performance and present a comprehensive experimental evaluation of SBCO algorithm using micro-benchmark, real-world concurrent and non-concurrent workloads. We first introduce our experiment environment and benchmarks selected, and then evaluate results and compare the performance of SBCO with the default scheduler as well as other representative solutions.

4.6.1 Experiment Design

We ran all experiments on Dell PowerEdge1950 physical machines with two quad-core Intel Xeon CPU and 8GB memory, running Linux kernel 2.6.34.4. The guest VMs run CentOS 5.4 without any modification, and unless specially stated, all VMs are configured with 4 $vCPUs$, thus each VM's $vCPU$ number is equal to the $pCPU$ number. Note that it is the vertical length of a $pCPU$ run queue that affects a VM's

scheduling distance, instead of the horizontal total number of run queues. Inside a guest VM, we selected following workloads to saturate vCPUs. Meanwhile, on a physical server, besides consolidating more than one VMs so that there are vCPUs processes in each run queue, we also ran CPU intensive workloads to saturate pCPUs to increase the size of run queues. The detailed specifications of the benchmarks we used to measure the SBCO's performance and overhead are listed as follows:

Parsec Parsec is a benchmark suite for Chip-Multiprocessors (CMPs) that focuses on emerging applications. It includes a diverse set of workloads from different domains such as interactive animation or systems applications that mimic large-scale commercial workloads [19]. We used the pthread implementation of the benchmarks which uses `spin-lock` for synchronization.

SuperPI SuperPI [5] is a CPU-bound workload to calculate the digits of PI. We run SuperPI in a few VM as CPU intensive workload and also use it as disturbance workload.

NPB NAS parallel benchmarks [17] contain 9 parallel programs derived from computational fluid dynamics applications. We activate the environment variable `OMP_WAIT_POLICY` to allow benchmarks using busy-waiting synchronization.

Kernbench We use the parallel make benchmark, Kernbench [3], to compile Linux 2.6.34.4 kernel source with 16 threads (`make -j 16`) and use the kernel compile completion time as the performance metric. The VMs running Kernbench are configured with enough memory to avoid swap storms.

SPECjbb We use SPECjbb2005 [4] v1.07 and BEA JRockit 6.0 JVM. It emulates a three tier client/server system by spawning multiple java threads to simulate users transaction requests in multiple warehouses. Synchronization is required when user

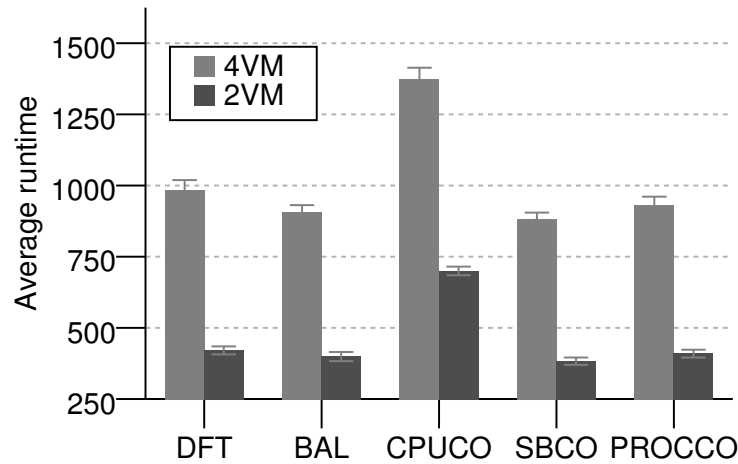


Figure 4.6: Average runtime of kernbench.

requests and server side management operations need to access the same database table. We start with one warehouse(thread) and stop at 16, and report the average business operations per second(bops) from 8 to 16 warehouses.

4.6.2 Experimental Results

Performance

We ran Kernbench in one 4-vCPU VM with other one or three VMs running CPU-intensive SuperPI workload and measure the completion time of KernBench. To avoid swap storms and eliminate uncertainties, we assigned about 2G memory to each VM to allow KernBench caches all the Linux kernel source in the RAM. KernBench frequently reads files or links through Linux VFS layer, thus incurring file system's inode lock contentions, which is protected by spin-lock in kernel space. We compare the completion time of KernBench due to following different scheduling approaches: the default CFS scheduler (*DFT*), balanced scheduler (*BAL*), process

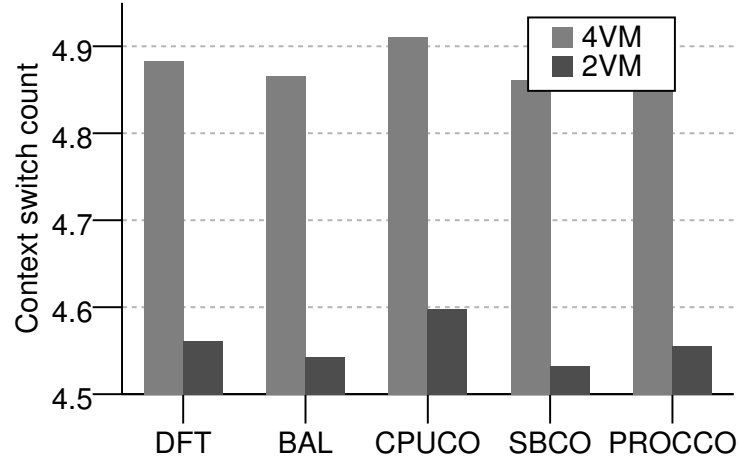


Figure 4.7: Context switches numbers of kernbench in Log scale.

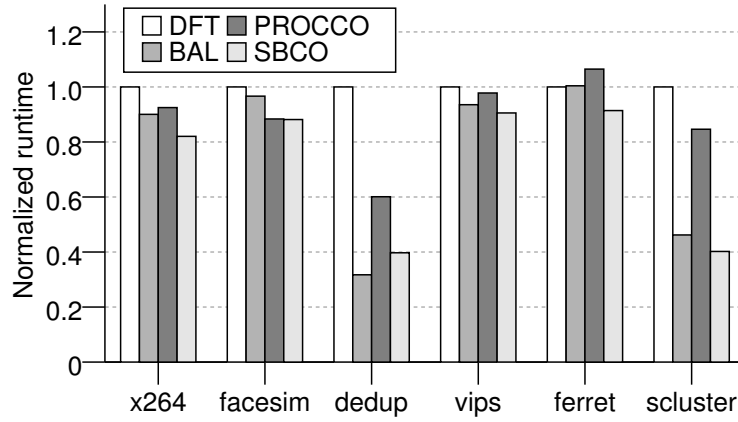


Figure 4.8: Parsec performance with average rq size is eight.

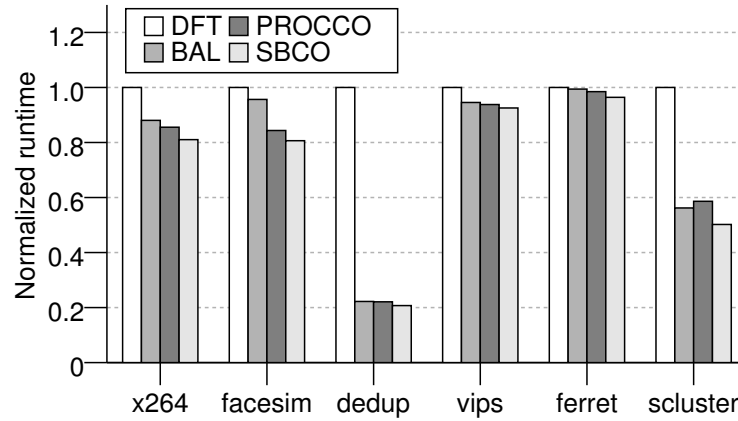


Figure 4.9: Parsec performance with average rq size is twelve.

based co-scheduling (*PROCCO*), CPU based co-scheduling (*CPUCO*) and our *SBCO*.

Figure 4.6 shows that, due to the heavy lock contention, the default CFS scheduler performs worst compared with *BAL* and *SBCO*, both of which split vCPUs to different run queues to reduce the overhead of LHP. *SBCO* achieves 14% performance improvement over *DFT* and 6% over *BAL*. From Figure 4.6, we also observe *CPUCO* leads to performance degradation significantly. This demonstrates that allow one pCPU to lead other pCPUs to co-schedule sibling vCPUs is not necessarily feasible as expected, thus we remove *CPUCO* for comparison in our remaining evaluation. Kerbench also provides the count number of context switches during the execution. Figure 4.7 shows *SBCO* is capable of reducing the number of context switches by at least 3%, that is equivalent to a large amount of context switches given *SBCO* cause as many as 76400 context switches. *CPUCO* leads to performance loss due to the tremendous increase of the number of context switches.

Figure 4.8 shows the normalized performance of Parsec benchmark due to

different scheduling approaches in one VM. We also ran three other CPU-intensive SuperPI VMs to make the average queue size of each pCPU length stays at eight. Though different workloads have different average runtimes, *SBCO* outperforms *DFT* as well as other approaches in all test cases. More specifically, for the *dedup and scluster* workloads, *SBCO* improves performance by up to 68% and 52% respectively compared with the *DFT* case. At the same time, *SBCO* outperforms *BAL* by 7% and 9% respectively. Note that the *dedup* benchmark uses a pipelined programming model to parallelize the compression to mimic real-world implementations [19]. Both *SBCO* and *BAL* avoids the LHP issue resulted from frequency synchronization between pipeline steps. Similarly, *scluster* gains benefit from *SBCO* while processing large amounts of continuously produced data. We also observed *BAL* performs closely to our *SBCO* with workloads such as *x264*, *facesim*, *ferret*. There are two reasons for such close performance improvement. First, *SBCO* is also built on distributing sibling vCPUs to different pCPUs, which is the core of *BAL*. Therefore, *SBCO* works like *BAL* unless there is large scheduling distance detected. Second, *SBCO* involves marginal additional cost to minimize the scheduling distance between sibling vCPUs(analyzed in 4.6.2). If the workload itself does not have large amount of synchronization between threads, the balancing only affects short-term fairness. In Figure Figure 4.9, we ran four more VMs running CPU-intensive workload so as to increase the average run queue size to be twelve. *dedup* achieved even higher performance gain (up to 70% over *DFT*) compared with its performance gain in four VMs case in Figure Figure 4.8. Such phenomenon demonstrates scheduling distance can contribute to significant performance loss when the average pCPU run queue size increases.

Throughput

To evaluate the effect of *SBCO* on applications' throughput, we kept one VM running SPECjbb benchmark and compared the average throughput due to different scheduling approaches. Meanwhile, we increase the number of disturbing VMs from one to five. All VMs was configured with four vCPUs, the same as the total number of pCPUs. Each disturbing VM ran the CPU-intensive Supper PI workload to keep pCPUs busy so as to maintain the same amount of average run queue size. These CPU-intensive applications usually keep occupying CPU resource and get preempted by the scheduler once they use up their time slices. Therefore the more the disturbing VMs, the longer the run queue, resulting in large waiting time due to large scheduling distance. We used one single JVM instance for SPECjbb benchmark and gradually increased SPECjbb workload by increasing the its warehouses numbers. The average throughput is shown in Figure Figure 4.10. It can be seen that *PROCCO*, *BAL* and *SBCO* outperform the default CFS due to their alleviation of the synchronization latency problem. *SBCO* achieves about 6% higher throughput than the *PROCCO* due to the reduction of context switching cost. It also yields 4% higher throughput compared with *BAL* because of the mitigation effect of scheduling distance. From Figure Figure 4.10, we can also observe *SBCO*'s performance gain is higher in the five disturbing VMs case compared with there is only one disturbing VM. It is because long run queue tends to incur relatively high synchronization delay.

Scalability

To study the scalability of *SBCO*, we ran different Parsec workloads in one VM and increased the average pCPU run queue size by launching more CPU-intensive

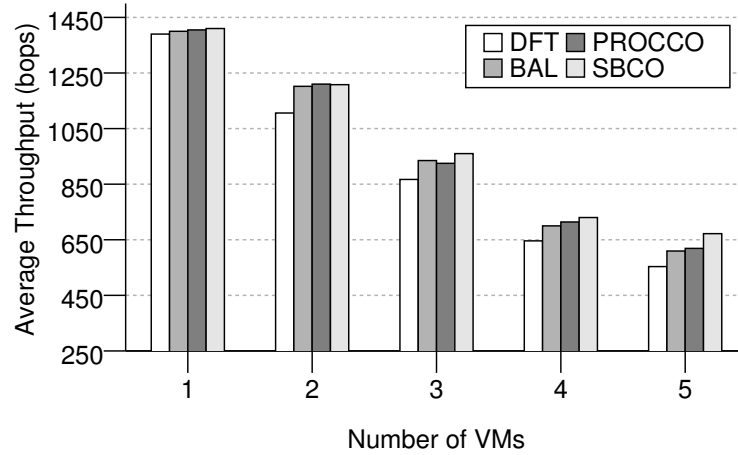


Figure 4.10: Performance of SPECjbb benchmark.

applications on the physical host. *SBCO* identifies a vCPU process by checking the name of a thread. It always dispatch sibling *qemu-kvm* processes to different run queues. In our experiment, we note that running a large amount of disturbing VMs requires huge physical memory space. Instead, we ran multiple four threads CPU-intensive applications and assign threads' name to be *qemu-kvm*. Therefore these disturbing threads are also treated like vCPUs and they are dispatched to different run queues. The average run queue size is increased gradually with more disturbing threads being launched. Figure Figure 4.11 shows the normalized completion time of different Parsec workloads with respect to the default *DFT*. As suggested by the normalized numbers in the figure, *SBCO* is able to improve the performance by 61% with *dedup*. The cost of iterating a vCPU's sibling vCPUs and calculating the scheduling distance remains unchanged in all the cases. *dedup* benefits from *SBCO* most due to its heavy synchronization overhead.

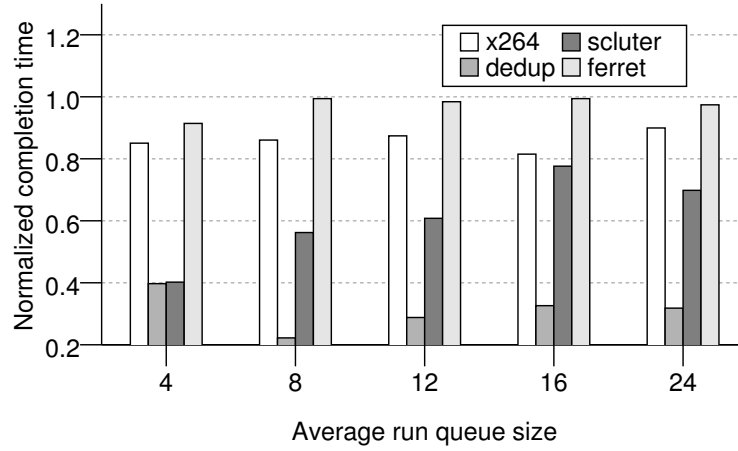


Figure 4.11: Scalability of SBCO on Parsec workloads.

Fairness

In this section, we show the effectiveness of SBCO in VM level fairness. We ran four VMs with multithreaded CPU intensive NPB workload to saturate vCPUs. On the physical host, we implemented a kernel thread to periodically sample the total execution time of each VM by summing up each vCPU's execution time. The sampling period varies from 1s, 5s, to 120s and each sample calculates the maximum difference, referred as lag, between VMs. The configurable sample period is open to user applications through Linux's *sysctl* interface, and the sampling thread is assigned with highest priority to avoid competing CPU resource with vCPUs. Let T_{vm} represent the sum of all vCPUs' execution time in a VM and Lag_t be the maximum difference of the execution time of all the VMs at time t , denoted as maximum absolute lag(MAL). We repeated the experiment for five times and present the average lag value in Figure Figure 4.12.

$$T_{vm} = \sum_1^n T_{vCPU_i}, n = 4;$$

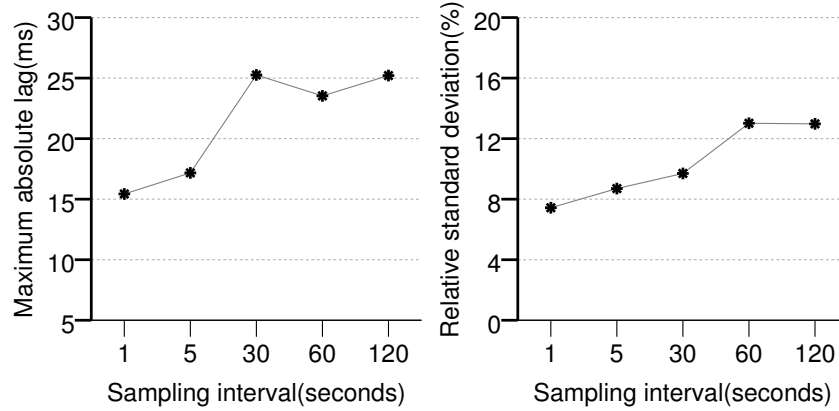


Figure 4.12: Relative Standard Deviation(RSD) of the Maximum Absolute Lag(MAL) of each VM.

$$Lag_t = Max(T_{vm_i}) - Min(T_{vm_j}), i, j \in [1, 4].$$

Recall that each VM's vCPUs are distributed into different run queues, thus each pCPU run queue only has one vCPU thread of every VM. Figure Figure 4.12 shows the MAL and the relative standard deviation(RSD) with respect to different sample intervals. As shown in Figure Figure 4.12, the lag varies a little with different sample periods. More specifically, when the period goes from 1s to 120s, the average maximum absolute lag varies from 15ms to 25ms. Compared with the average 20ms maximum lag with default CFS scheduler, our SBCO has a negligible impact on scheduling fairness between VMs. According to the RSD, the variation ranges from 7% to around 14% and the overall RSDs are bounded to 15%.

4.7 Summary

In this work, we propose *SBCO*, a new scheduling scheme for performance optimization in virtualized SMP environment. *SBCO* first inherits the advantages of

traditional co-scheduling such as minimizing synchronization latency and speedup the communication between vCPUs. Meanwhile, it avoids the scheduling fragmentation and priority inversion issue because *SBCO* does not demand co-scheduling all the sibling vCPUs precisely at the same time. Instead, it coarsely adjusts the sibling vCPUs position in their respective run queues for balance purpose and facilitate sibling vCPUs to be scheduled coarsely at the same level. In other words, *SBCO* dynamically adjusts the affinity of vCPUs to avoid sibling vCPUs to exist in the same run queue, like the previously proposed *balance scheduling* algorithm. It also balances the sibling vCPUs in the different run queues. We implemented the prototype of *SBCO* based on CFS scheduler and conducted evaluations with KVM VM. Our experimental results show that *SBCO* brings more than 10% performance improvement for many applications.

The LHP may have different effects on distinct applications, depending on the usage of spin-lock for synchronization. The impact of scheduling distance may also vary with the characteristics of applications. In the future, we plan to further study applications' sensitivity to the scheduling distance of sibling vCPUs and propose an online adaptive threshold for the purpose of dynamically balance sibling vCPUs with different granularity. If such threshold is restricted to zero, then all the sibling vCPUs should be expected to run at the same time, and *SBCO* reduces itself to be the conventional strict co-start and co-stop co-scheduling.

Chapter 5: FLEXIBLE MOBILE AUGMENTATION

5.1 Introduction

Mobile devices, such as smartphones or tablets are getting more and more popular. Meanwhile, many of today's smartphones, with full sized screens, advanced features like camera(s), GPS and accelerometers, are competing with existing laptops and desktops for the market popularity. At the same time, mobile application developers are building even more complex applications, such as gaming, video editing, augmented reality, navigation, and speech recognition, which are used to be only on PCs and require considerable computing power and energy. These applications greatly extend the functionalities of mobile devices and provide excellent mobility and user experiences. However, these applications also pose new challenges to the hardware computing capability, storage space and battery life. Many mobile devices have significant limitations imposed upon them due to the desirability of portable sizes, lower weights, longer battery life and other features. This often severely constrains software and hardware developments for these mobile devices.

The combination of cloud resources with mobile computation is an appealing solution for augmenting the computing capability of mobile devices and improve user experience. There are a few ways of using cloud computing for mobile phones. First, the computation can be partially processed outside mobile devices [38, 24, 70]. In this case, a mobile application can be split in the traditional client-server paradigm and lets the resource-intensive tasks be performed in the cloud syncing the results back to mobile devices. Such execution offloading turns mobile devices to be simply thin clients accessing abundant computing resources in cloud, providing a convenient

way of boosting mobile applications' performances. Another popular approach of integrating cloud is that the computation is carried out on mobile devices, while cloud serves as an unlimited storage server. Additionally, another mobile cloud model could be leveraging the cloud to process data [61]. In an in-cloud antivirus system, mobile devices can send the suspicious files to the antivirus service in the cloud for scanning to avoid performing resource consuming scanning applications locally on a phone. Among all these models, mobile computation offloading usually requires application developers to manually or programmatically separate the resource intensive functions or methods of the application from the source code. The possible partitioning also varies with computing environments. For example, partitioning on low-end mobile devices with intermittent connectivity may not be optimal for high-end devices with good connectivity.

To avoid the cumbersome application partitioning, a few recent research works [99, 24] proposed to use VM to provide a clone for computing environments of mobile phones. The clones are usually hosted in a VM, which running on a x86 server that emulates ARM architecture. However, ARM is known for excellent power consumption, and compact code. Recent CPUs based on the ARM architecture are also quite powerful, and are being incorporated into a wide range of products including both mobile devices and servers. Recent advancements in software and hardware facilitate the virtualization on ARM architectures. Mobile users today are able to run multiple phone instances on the same mobile device at the same time. Since, ARM-based server has been a new option for the classic x86 server for deploying data-centers [12], ARM-based cloud can run all the mobile applications without emulation and the computing capability gap between mobile devices and cloud can be minimized by

running the virtual phone instance in ARM-based cloud.

In this work, we proposed a new framework to maintain an augmented virtual mobile phone clone in the ARM-based cloud. Such augmented clone has more computing power, more memory and disk space, which contains the real phone as a subset. Meanwhile, mobile devices run virtualized phone instances to isolate the computing environments. The time consuming or resource hungry applications are shifted to run remotely in the clone in cloud and synchronize the results back to the phone instance on the phone. The input events from keyboard, touch screen are recorded and then sent to the augmented clone. All of users interactive events are replayed with deterministic reply and all the disk changes are incrementally synchronized with the phone instance on mobile devices. In this communication model, we are able to significantly increase the computing power of mobile devices and reduce the energy consumption. Additionally, this model allows mobile users to quickly exchange data in the cloud over a high speed network.

5.2 Background and Challenges

In this section, we first introduce a few mobile cloud computing models. Then we discuss the differences between classic server cloud and mobile cloud and elaborate some of the challenges of achieving mobile cloud computing.

5.2.1 Mobile Computing and Cloud

The early definition of mobile cloud computing lies on the basic mobile plus cloud approach. In this model, cloud, usually equipped with abundant hardware resources runs applications to provide certain type of services such as email or storage services. Mobile devices play a role of a cloud access terminal or a thin client. The inborn

mobility of mobile devices greatly increases the accessibility of cloud services. In addition, to this "mobile access to cloud" model, another form of mobile cloud uses each mobile devices as collective sensing components in a mobile network. Each component acts like a data collector node, collecting environmental metrics or location and reporting back to the cloud. Mobile devices in this case act as peripheral device of the cloud. Similar to this cooperative data collection, mobile devices can also be configured as one computing unit, as referred as crowd computing. For example, a mobile cluster can be formed with a few low energy consumption mobile devices and configured as a Hadoop cluster for big data processing.

Recent research Cloudlets[79] proposed another way to integrating cloud resources. Cloudlets are defined as decentralized and widely-dispersed internet infrastructure whose computing cycles and storage resources can be leveraged by nearby mobile devices. In this model, mobile devices offload their workloads to a local cloudlet, which has the connectivity to the remote cloud servers. The cloudlets usually have the same general architecture as a normal computer, but are smaller, less powerful, less power hungry and less expensive. These cloudlets could be installed in common areas such as coffee shops so that mobile devices can act like a thin client to connect to the cloudlet rather than directly to a remote cloud server which has bandwidth and latency issues. CloneCloud[24] proposed similar offloading approach to seamlessly leverage cloud to execute resource expensive applications. It augments the capabilities of smartphones by moving, in whole or in part, the execution of the applications to cloud. The partition resource expensive applications is determined by a static program analyzer followed by dynamic program profiling. The mobile device clone is the duplication of the whole mobile device.

The classic server virtual machine consolidation and live migration greatly increases hardware resource utilization and allows cloud users to pay for cloud resource on the go. None of the above models involves ARM based mobile virtualization. Mobile cloud computing should differ from simple computation offloading in the sense that the cloud can offer services other than computing for mobile clients. In this work, we explored the possibility of leveraging mobile virtualization to build ARM based mobile cloud. In the remaining section, we discuss a few challenges of mobile cloud.

5.2.2 Architecture Compatibility

Server virtualization technology exists for years and has been mature. However, virtualization on mobile devices has been evolving relatively slow. Hardware vendors are gradually embracing virtualization and developing new features to simplify virtualization technology. They enhanced hardware chips by providing hardware-assisted x86 virtualization technologies such as Intel VT19 and AMD-VTM20. These technologies allow virtual machine monitor(VMM) efficiently virtualize all the instruction sets by handling sensitive instructions using a classic trap-and-emulate model running at privileged level. Besides the hardware assisted instruction trapping, modern x86 CPU also have a memory management unit(MMU) and translation looked buffer(TLB) to coordinate and optimize virtual memory management in VMM. Due to the majority of today's mobile devices are using reduced instruction set computer (RISC) architecture like ARM, ARM chip vendors started to support virtualization in recent years. ARM v7-A introduced ARM Virtualization Extensions and System Memory Management Unit(SMMU) architecture. These ARM virtualization extensions allow for a new hypervisor execution mode and enables the VMM to run at

a higher privilege level than the guest mobile OS. It also provides the mechanisms to simplify interrupt handling. The SMMU supports multiple page translation contexts and two levels of address translation as well as hardware acceleration. These hardware features facilitates the shifting of virtualization from servers to mobile devices. However, majority of cloud infrastructure today are still built with Complex Instruction Set Computer architecture. Since running mobile application in the cloud or partially offloading certain functions to the cloud requires the execution the same instruction set. This architecture compatibility issue limits the integration of cloud with mobile devices. Moreover, mobile devices are usually shipped with hardware components such as wifi, camera, GPS and various sensors. Thus traditional server VM consolidation or live migrations between mobile device and cloud tends to be hard to achieve. Classic computation offloading loses the strength when interacting with all these low-level hardware components.

5.2.3 Cost of Communication and Computing

Due to the limited computing power and battery life, local computing on mobile devices usually have lower performance and shorter battery time compare with involving remote execution in the cloud. However, mobile cloud incurs additional cost of communication and also requires strong internet accessibility. Thus, the balance between local and remote computing is a trade-off commonly between communication cost and computation gain. For certain tasks that do not need to be performed immediately, such as virus checking or indexing files, the communication cost involved by moving them to the cloud is far smaller than the performance improvement that will gain. On the other hand, there are some tasks whose computing intensive parts could be separated from less intensive parts. Thus the former parts, such as speech

recognition or video indexing, can be promoted to the cloud while leaving less intensive tasks to still be executed on the mobile devices. In this case, performance gain still outperform the communication cost. In contrast, some interactive applications require only minimal computing power but may incur significant communication cost if they are moved to the cloud. The remote execution in cloud needs to be carefully designed in order to avoid introducing long latencies which impacts the interactive user experience.

The communication pattern between mobile devices and cloud is another contributing factor to the performance and impacts the feasibility of remote execution. Due to data transferring through Wifi or 3G consumes unneglectable amount of the battery life, the granularity of the communication directly decides the communication cost. Interactive applications which usually requires frequent data exchange between mobile devices and cloud could use bulk data transfers. If immediate response is not required, the data transfer can be accumulated and postponed to deliver in parallel with other data later. This type of communication pattern is often application dependent. Consequently, the remote execution model is only preferable for certain applications that the cost and computing gain is balanced. In general, the underlying assumption for leveraging cloud to augment mobile devices lies in the assumption that it is worthy of integrating cloud as long as the performance of mobile devices can significantly improved or the mobile application can run more reliable or secure in the cloud. In most cases, such balance or worthiness is application dependent. In this work, we explored a general approach of using cloud resources.

5.3 System Design

Though today's mobile devices has been improved in both CPU frequency and memory space significantly in recent years, the computing requirements of mobile users, especially enterprise users, is still not fully achieved. A few intrinsic limitations of mobile devices hinder the feasibility of intense mobile computing and motivate the natural integration of cloud. In this section, we discuss the design of our AM-Phone and introduce the system components as well as the communication mechanism between these components.

5.3.1 Overview

Running a mobile phone virtualized clone in the cloud is conducive to conserve the scarce battery life and overcome the limitation due to the memory size or CPU power which leads to better application performance. Such mobile cloud computing model can significantly reduce on-device resource consumption. Our AMPhone does not require any modification on the applications. The clone in cloud and the real physical device can run identical binaries. Consequently, mobile applications are not constrained by the computing capabilities of mobile devices and can be configured with multiple CPU cores or big memory space. In essence, traditional computing on mobile devices is transformed into a distributed execution in cloud powered by high speed network connection and high processing capability. On the other hand, the whole mobile system replication releases programmers from manually or pragmatically partition the applications into the parts that run on the mobile phone and the parts that will run in the cloud. Such partitioning requires programmer to pay extra attention to the resource(CPU or memory) intensive pieces of code and limits the

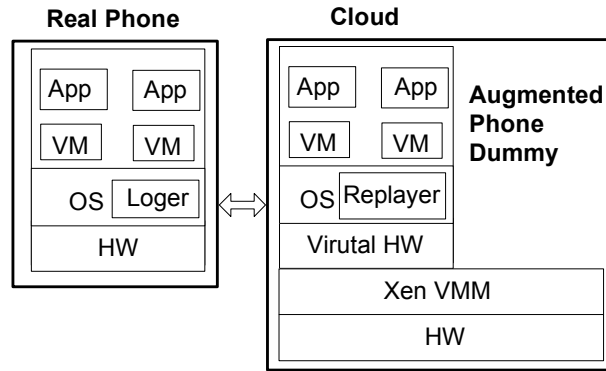


Figure 5.1: System Architecture.

flexibility of creating mobile applications. In addition, mobile users have the option to isolate their personal device and corporate device and have better privacy and security protection. For instance, malicious software or virus detection, which are unlikely feasible on mobile devices due to the limitations of hardware and battery resources, could be conducted in cloud efficiently and end users do not need to maintain or update their virus database. Corporate also can have better protection over their business related information.

5.3.2 Whole System Replication

One of the design goals of our AMPhone is to avoid traditional application partitioning. It is time inefficient and challenging to require programmers to isolate all the potentially resource intensive chunks of source code from those will be ran on mobile devices. Programmatic partitioning seems less cumbersome, but unlikely to produce highly optimized source code. As a result, AMPhone is designed with whole system replication. As shown in Figure 5.2, each mobile device runs one or more isolated virtual phone instances. In the cloud, the servers have the same ARM architecture as mobile devices and host a virtual phone instances pool. As shown in Figure 5.1, in-

stead of creating thread level VM for an application, we wrap the entire environment and deploy an additional copy in cloud to run the same application binary. Thus there is no need of partitioning applications.

5.3.3 Mobile Augmentation

Mobile computation augmentation enables mobile devices to increase, enhance, and optimize computing capabilities by leveraging various software or hardware approaches. Hardware approaches include improving the capability of physical components, such as CPU, memory, storage, and battery. Software approaches contain computation offloading, remote data storage, remote execution, etc. Mobile augmentation is able to increase computing capabilities of mobile devices and conserve energy, especially for computing-intensive applications. Our AMPhone inherits the nature scale-on-demand feature of cloud computing. In our design, a virtual phone instance in cloud can be configured with more CPU cores, more memory or storage space upon the applications' demand. Unlike classic application level computation offloading or remote execution, these virtual phone instances encapsulate all the applications running on the mobile phone and boost their execution capability at the same time.

5.3.4 Incremental Synchronization

In the process of augmentation, the native applications on mobile devices need to synchronize their data with the cloud to ensure the consistency and integrity between mobile device and the cloud. The communication cost of such synchronization can counteract the performance gain of the cloud. The data synchronization can significantly increase the communication traffic and hurt the execution time and energy efficiency, especially for data or communication intensive applications, where frequent

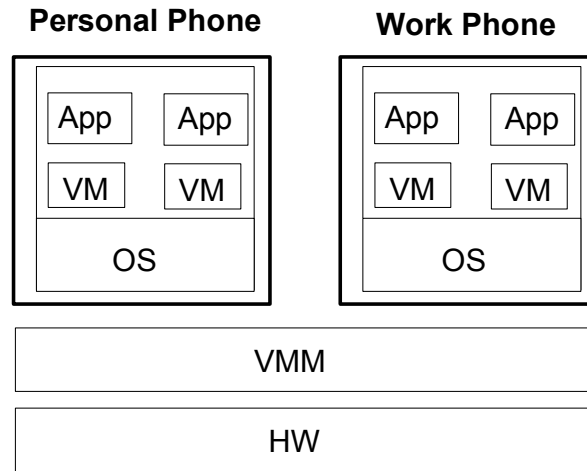


Figure 5.2: Virtual Phone.

synchronization is desired. To address this dilemma, we designed AMPhone to incrementally synchronize the data changes between mobile devices and cloud. We first capture the keyboard input on the actual mobile devices, accumulate the keystroke events and then send to cloud to reply these events on the virtual phone instance. After the execution of virtual mobile instance, storage changes are accumulated and incrementally synchronized to mobile devices in a batch. Complex applications that access large amount of data can just synchronize the execution results back to mobile devices and leave the input data in cloud. This flexibility also simplifies the data sharing between mobile users.

Another design option is to lively migrate mobile VMs between mobile phone and cloud. The live migration technology has been widely adopted in server virtualization to lively reorganize VM cluster for maintenance or for updating the scale of the cluster. As a result, it can also reduce service downtime while seamlessly moving the VMs between hosts. However, live migration usually requires the destination host and the original host share the same centralized storage servers to avoid moving large volume

data. Considering the wireless network bandwidth, intermittency, and the fact that most mobile devices are not offering backend services that may expect strict service down time, live migration may simply increase the cost and defeat the benefits of integrating cloud. In this work, we design AMPhone to use the classic suspend-and-resume model. The mobile clones doesn't share the data storage with mobile devices. Once the mobile os is suspended, the system states are check-pointed and all the changes made from last checkpoint are synchronize to cloud and user input are recorded and replayed in the cloud. To trigger a checkpoint, the application notifies the local VMM to identify and record the input events.

5.3.5 Limitations

Mobile devices today are no longer just another type of computing unit. They are usually equipped with GPS, camera or various sensors. All of these low-level hardwares pose challenges to mobile virtualization and mobile cloud computing. The applications in a virtual phone clone that have to access the physical hardware, for example an application needs to access GPS location or an application requires blue-tooth interface communication, then have to frequently synchronize the low-level input with the cloud clone. Due to the communication cost, these types of application may not be the idea applications that can benefit from running the virtual clone. In addition, since all the information on the phone including user's applications and personal data has a duplication in cloud, this complicate virtual phone instances management, especially when user data security protection and data privacy have to be enforced. Cloud providers may have to take the responsibility to isolate a user's data from others and secure all the access to the data in cloud. Since data security and data protection are independent research topics of cloud computing, our prototype

design only considers the data security during transferring data.

5.4 Implementation

We implemented AMPhone prototype on Xen 4.4 and a development board powered with AllWinner A20 chip, which is ARM Cortex-A7 32bits RISC CPU. The A20 processor is an SoC with dual Cortex A7 CPU cores which delivers decent computing capability while consuming less power. It also integrates the Mali400 MP2 GPU and supports the ARM virtualization extensions. A20 is one of the ARM CPUs with hardware virtualization support. On mobile devices, we added a kernel module to monitor the key stroke event and repeatedly record the events. In the backend cloud, we changed the VMM to correspond the data synchronization from mobile devices and control the running status of the augmented clone.

Event Monitoring and Reply In order to capture the input from mobile devices, we implemented a kernel module to keep tracking all the keystroke event on mobile devices, buffer the stream of events and then forward the events to cloud. This module contains two components. One is running on mobile devices to constantly capture user's input. When mobile device is suspended, only the input is captured and forwarded, the mobile stops running any applications or changes the files on the storage device. The other module is running in the virtualized clone after the clone is resumed in cloud. The accumulated events from the mobile devices are replayed in the virtualized clone. To reproduce a deterministic replay of interleaved execution of native applications on mobile devices and the execution of virtualized phone, we enforce that the virtual clone in the cloud is activated after the mobile OS is suspended and the data changes are synchronized to cloud.

Incremental Synchronization To keep the mobile phone always synchronized with the virtual clone in cloud, we implemented a daemon to constantly monitor the changes of the mobile phone and update the changes with the cloud. On the other hand, when the mobile clone is updated, the changes have to be pushed to the mobile side. To simply the data transfer under different wireless connections, we assume the mobile phone is connected to the cloud through wifi and the data change is synchronized in mutual direction with very fine granularity. Due to the typical storage space of mobile device ranges from 16GB to 64GB and the fact that most of the mobile applications are designed to run inside a sandbox which means each application can only update the files or folders within the storage space owned by that application, the total maximal synchronization size is very limited. One optimization of current implementation is to adjust the synchronization frequency by setting a threshold for the size of changes to trigger the synchronization.

5.5 Evaluation

In this section, we first describe our experimental setup and our benchmarks. Then we present some experimental results and our observations.

5.5.1 Experimental Setup

Mobile device performance measurements were obtained using Cubieboard2 [11] with a dual core 1GHz CPU on a AllWinner A20 SoC. We use Cubieboard2 [11] to run two virtualized mobile phone instances and use Dell PowerEdge1950 which has two quad-core Intel Xeon CPU and 8GB memory to simulate the ARM server [12]. Based on the configuration of one of the representative ARM server [12], we configure the virtual VMs in cloud with the same CPU and memory resource as the

ARM server. The virtual phone instances on Cubieboard2 are connected to cloud through WiFi. We present some experimental results that quantify the performance of a few benchmarks on both mobile devices and cloud. We evaluate the virtualization overhead of running two virtualized phone instances comparing to running one instance with native execution with micro-benchmarks. We measure and compare the performance of applications running in virtualized phone and cloud to demonstrate the feasibility of flexible mobile computing augmentation. Due to the differences in hardware architecture and the simulation of ARM server, we present the relative performance difference between virtualized instance on mobile device and cloud instead of the absolute application performance. Following are the benchmarks we used in the evaluations.

lmbench micro-benchmarks. lmbench [54] is a portable micro-benchmark suite designed to measure important aspects of system performance such as process creation cost, context switching cost, single handling cost etc. Our AMPhone proposes to run multiple phone instances on the same mobile device via mobile virtualization. We use the lmbench benchmarking suite to measure overheads of virtualization on mobile devices and compare these various basic cost on mobile devices with that in cloud.

Octane Octane [31] is a benchmark suite that measures a JavaScript engine's performance by running a suite of tests representing today's complex and demanding web applications. Octane is designed to measure the performance of JavaScript code found in large, real-world web applications, running on modern mobile and desktop browsers. Due to a large amount of objects like flash or javascript code embedded in modern web sites, mobile Web page loads are usually very slow. In our evaluation, we measure the performance comparison between rendering websites in virtualized

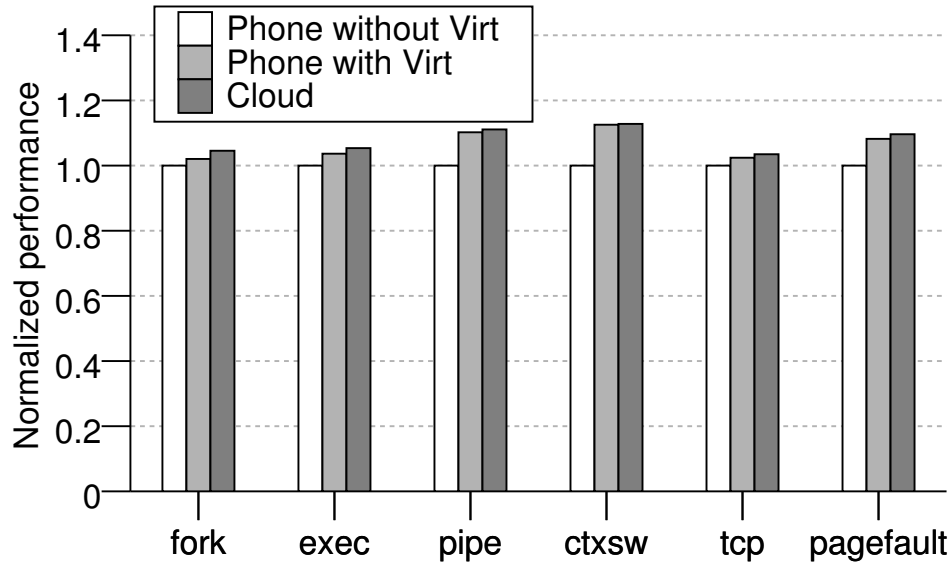


Figure 5.3: lmbench performance.

mobile instance on mobile device and augmented instance in cloud.

Openssl speed The OpenSSL project provides an open source implementation of the SSL/TLS protocols, and is widely deployed on mobile devices and servers. The SSL/TLS protocols have two phases: an initial session-initiation/handshake phase, and a bulk data transfer phase. The session initiation cost directly tie to applications' performance. We measured the performance of OpenSSL using the built-in speed test tool [63] with different algorithms. Each test was run a few times and an average was taken for each data point.

5.5.2 Evaluation Results

We first measure the performance impact of mobile virtualization with lmbench by evaluating the slowdown of some primitive operating system operations. Figure 5.3 shows the normalized performance for running lmbench in a virtual phone

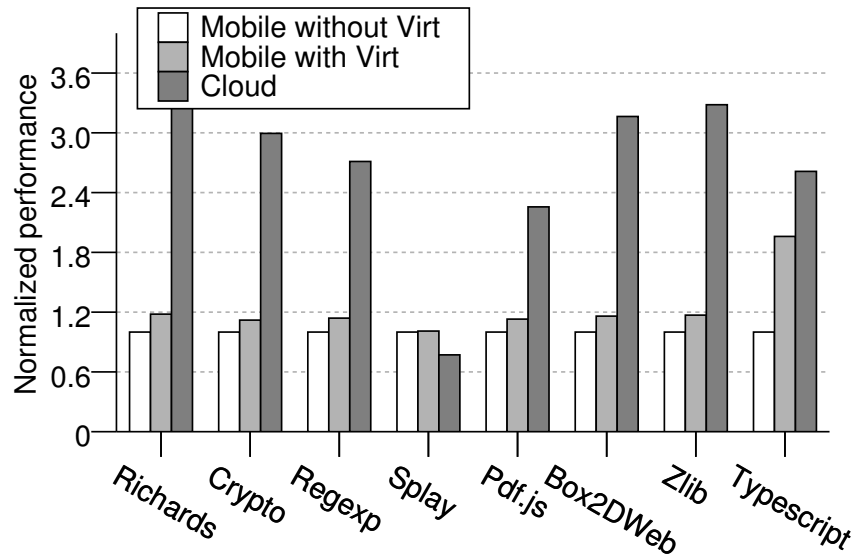


Figure 5.4: Octane performance.

instance versus running directly on a mobile device. For comparison, we also show the performance of running lmbench in a VM in simulated ARM cloud. From the figure, virtualization incurs from 5% to 8% additional overhead for fork, exec and tcp benchmarks. In addition, according to Figure 5.3, the overhead in cloud is higher than that in a virtualized phone instance. This is due to server in cloud has higher IPI overhead than ARM architecture and both context and pipe operations involve repeatedly sending IPI. Although these individual operations cost more in cloud, it doesn't contradict the purpose of mobile augmentation because cloud usually has much more hardware resources and mobile devices. In the remaining section, we evaluate the performance improvement with browser related benchmarks and real applications.

Figure 5.4 shows the the performance comparison of running Octane benchmark

in all cases. For testing web browsing performance, we used Chrome to run the Octane Javascript benchmark. Cloud VM running Octane received 5400 overall score and the mobile device without virtualization had only 917 score, nearly one fifth of the performance in cloud. Such contrast is also reflected by different benchmarks. As shown in the figure, cloud VM has ranging from five to ten times better performance than the default no VM on mobile device case. Richards benchmark is an OS kernel simulation benchmark focusing on property load and store, function or method calls. This benchmark got 10 time performance improvement by leveraging more powerful CPU in cloud. Considering mobile web browsing is one of the main activities happening on mobile devices and also one of activities that drains battery life. According to the evaluation results, besides the suspend and resume model used by AMPhone, current AMPhone model could also be extended for browser specific applications. In the extended model, mobile browsers can selectively execute portions of the page loading process in cloud. In this way, the time consuming javascript loading and page rendering can be conducted in cloud first and only the final html pages are sent to mobile phone.

Figure 5.5 shows the the normalized performance comparison of running OpenSSL speed test with different algorithms. Y-axis represents the log scale of performance speed up. According to the figure, virtualization layer only incurs less than 5% percent performance degradation comparing with none-virtualized case. In contrast to the potential performance gain due to running applications in cloud, this performance impact is marginal. DSA 512 bits signature verification algorithm can boost performance 12x than running on mobile devices. This is mainly because this algorithm requires both memory and CPU. Running OpenSSL in cloud can generally improve

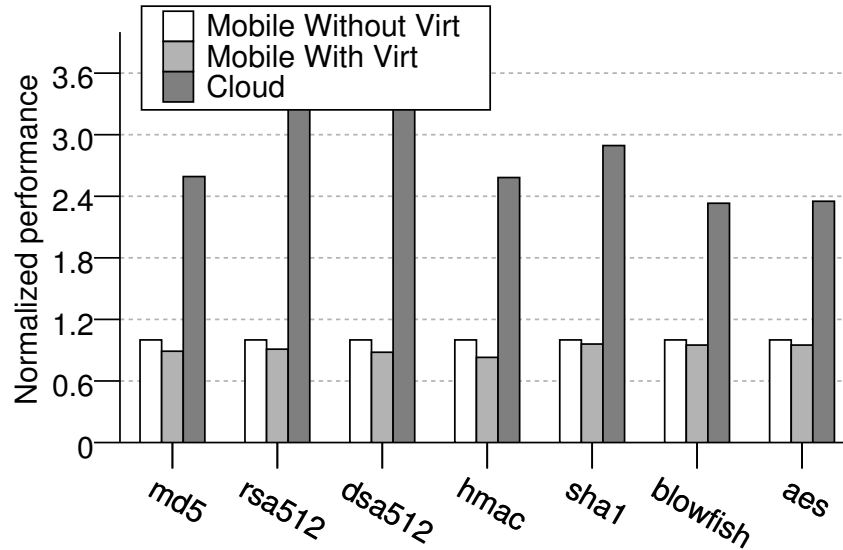


Figure 5.5: Openssl performance.

performance from five to twelve times.

5.6 Discussion and Summary

Due to the outstanding performance and low energy consumption, ARM processors have been dominating the mobile devices. Recently, there is also a growing trend that ARM processors are becoming a strong competitor to x86 processors on servers. At the same time, ARM processors have gradually started to support virtualization and classic server virtualization technology has been adopted on mobile devices recently. Our proposal envisions the future of mobile cloud computing with the confluence of the advancement of mobile hardware and mobile software virtualization. In this vision, mobile devices run more than one virtualized phone instances and are connect to ARM based cloud servers powered with abundant CPU cores, memory and storage space. The virtual instances on mobile devices can be seamlessly moved

to run in the ARM based cloud. In this design, service providers such as AT&T or Verizon can offer the cloud infrastructure to mobile phone users. Given the high speed wireless connection available on mobile phone, mobile users can dynamically run their applications in augmented clones without the manual efforts of application partitioning.

Mobile devices augmentation by leveraging cloud infrastructures has been an emerging research topic. The ultimate motivation of mobile augmentation is to boost applications' performance, save power consumption, and break the restriction of the limited resources on mobile devices. In this paper, we described the challenges of current mobile cloud computing and presented a new model for augmenting the capability of mobile devices. Comparing with most existing mobile cloud solutions, our system proposed a new mobile augmentation model to run virtualized phone instances on both mobile devices and cloud. In this design, mobile devices run multiple phone instances to provide isolated computing environment and each virtualized instance is also connected to an augmented duplicated phone clone running in cloud. Mobile applications runs in the virtualized clone in cloud and synchronize the changes back to the instance on mobile phone. This model offers users flexible augmented remote execution without the requirement of application partitioning. Our prototype experiment results show that this model is capable of both improving applications' performance.

Our AMPhone prototype provides the basic framework of augmented mobile cloud computing. Besides the performance improvement, AMPhone also provides different isolate phone instances, for example, users' personal phone instance and corporate phone instance are separated. Thus important corporate data could also be separated

from personal data and be protected with additional access rules. The current prototype hasn't encrypted the event message transferring between a local phone instance and a remote virtualized phone instance in the cloud. This might cause some security issues for applications that require strong security protection. In addition, all the meta data could be compressed before synchronizing to cloud. The compression minimizes the footprints and saves the power consumption during transferring.

Chapter 6: CONCLUSIONS

This dissertation aims to build agile mobile cloud system. In this chapter, we summarize our approaches presented in this dissertation and give the directions for potential future work.

6.1 Conclusions

In this dissertation, we have demonstrated that cloud resources can be used to boost the mobile computing capability and secure computing isolations. We introduced three building blocks of future mobile cloud computing: agile virtual phone clone deployment, efficient resource management in the cloud and flexible mobile augmentation. Under this vision, mobile users runs multiple virtualized phone instances on their mobile devices to isolate their computing environments. Each virtual instance also has an associated augmented clone in the cloud.

To effectively leverage cloud resources to augment mobile computing, we first analyzed the cost of each step in the process of VM deployment, and then we introduced the primitive of retrofitting VM deployment by using VM substrate to manage VMs in agile virtualized environment. We then presented pool based substrate management mechanism to efficiently manage virtual clones. With VM substrate, statefull VMs or VM clusters can be instantiated within sub-seconds and a virtualized phone clone can be also deployed in cloud on demand.

To effectively scheduling co-hosted virtual instances, we studied the classic LHP issue among consolidated virtual machines. We then proposed *SBCO*, a new scheduling scheme to improve performance and resource utilization in virtualized SMP environment. In particular, we optimize the CPU scheduling in the cases that many VMs are

consolidated on the same physical machine. With SBCO, many consolidated virtual phone instances can be efficiently scheduled in cloud.

With Substrate and SBCO, cloud side resources can be efficiently managed for mobile computing. Additionally, we proposed the AMPhone as a new mobile cloud model. In this model, virtualization technology is used on both mobile devices and cloud. A virtualized phone instance can be seamlessly suspended and resumed in the remote cloud where ample hardware resource is available. This model frees mobile devices from the limitation of computing capability. We presented the design, implementation and evaluations for each of those building blocks in this dissertation.

6.2 Future Directions

Mobile virtualization and mobile cloud computing have been very hot research topics recently. Along the line of this dissertation, there are a few other interesting issues and new directions deserve future exploring. In this dissertation work, we assume one to one model which implies each virtual phone instance on mobile device is associated to one augmented instance in cloud. In reality, for certain applications, it is possible to scale up the processing power by launching more than virtual instances in cloud for one task. This one to many model could further utilize cloud resources and enable desktop application models such as MapReduce and other parallel programs.

There is another type of mobile and cloud interaction mechanism, VM migration, could be further explored. In this dissertation, although we explained mobile devices are usually not acting as a backend server to serve incoming requests and service down time should not be crucial to mobile applications, the downtime of not responsible user interface still directly impact mobile user experience. Mobile virtual VM migra-

tion might still be an option for backend tasks requires long execution and less user interactions. Thus the feasibility and potential optimization of classic live migration deserves future tuning for mobile cloud case.

REFERENCES

- [1] *EC2 instance type*. <http://aws.amazon.com/ec2/instance-types/>.
- [2] *Inside the Linux 2.6 Completely Fair Scheduler*. <http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>.
- [3] *Kernbench Benchmark*. <http://freecode.com/projects/kernbench>.
- [4] *SPECjbb*. <http://www.spec.org/jbb2005/>.
- [5] *SuperPI*. <http://www.superpi.net/>.
- [6] *The VMware ESX Server*. <http://www.vmware.com/products/esx/>.
- [7] *lmbench*. <http://lmbench.sourceforge.net/>, 2001.
- [8] *Support Pause Filter in AMD processors*. <https://patchwork.kernel.org/patch/48624/>, 2001.
- [9] Intel 64 and ia-32 architectures software developer's manual volume 3., December 2011.
- [10] S. Abolfazli, Z. Sanaei, A. Gani, and M. Shiraz. Momcc: Market-oriented architecture for mobile cloud computing based on service oriented architecture. *CoRR*, abs/1206.6209, 2012.
- [11] AllWinner. http://linux-sunxi.org/Cubietech_Cubieboard2.
- [12] AMD. <http://www.amd.com/en-us/press-releases/Pages/64-bit-developer-kit-2014jul30.aspx>.

- [13] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10:95–109, 1992.
- [14] Apache thread pool. <http://commons.apache.org/sandbox/threadpool>.
- [15] A. Athan and D. Duchamp. Agent-mediated message passing for constrained environments. In *Mobile & Location-Independent Computing Symposium on Mobile & Location-Independent Computing Symposium*, MLCS, pages 9–9, Berkeley, CA, USA, 1993. USENIX Association.
- [16] Y. Bai, C. Xu, and Z. Li. Task-aware based co-scheduling for virtual machine system. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 181–188, New York, NY, USA, 2010.
- [17] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrisnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991.
- [18] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, pages 273–286, New York, NY, USA, 2003. ACM.
- [19] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th interna-*

- tional conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008.
- [20] T. Bird. Measuring function duration with ftrace. In *in Ottawa Linux Symposium*, 2009.
- [21] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE*, 2007.
- [22] R. C. Chiang, J. Hwang, H. H. Huang, and T. Wood. Matrix: Achieving predictable virtual machine performance in the clouds. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 45–56, Philadelphia, PA, June 2014. USENIX Association.
- [23] G. S. Choi, J.-H. Kim, D. Ersoz, A. B. Yoo, and C. R. Das. Coscheduling in clusters: Is it a viable alternative? In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing, SC '04*, pages 16–, Washington, DC, USA, 2004. IEEE Computer Society.
- [24] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *HotOS*, 2009.
- [25] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.
- [26] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *NSDI*, 2008.
- [27] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In

- 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 73–84, Philadelphia, PA, June 2014. USENIX Association.
- [28] Y. Dong, X. Zheng, X. Zhang, J. Dai, J. Li, X. Li, G. Zhai, and H. Guan. Improving virtualization performance and scalability with advanced hardware accelerations. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10, dec. 2010.
- [29] EC2. <http://aws.amazon.com/ec2>.
- [30] M. Gooderum, D. Mason, and J. Wrabetz. An integrated remote execution system for a heterogenous computer network environment, Oct. 14 1993. WO Patent App. PCT/US1993/003,106.
- [31] Google. <https://developers.google.com/octane/>.
- [32] S. Govindan, J. Choi, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: Communication-aware cpu management in consolidated xen-based hosting platforms. Jan 2009.
- [33] N. Har’El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky. Efficient and scalable paravirtual i/o system. In *USENIX Annual Technical Conference (ATC)*, 2013.
- [34] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. *SIGPLAN Not.*, 45(3):117–128, Mar. 2010.
- [35] J. Kim, R. A. Baratto, and J. Nieh. pthinc: A thin-client architecture for mobile wireless web. In *Proceedings of the 15th International Conference on World Wide Web, WWW ’06*, pages 143–152, New York, NY, USA, 2006. ACM.

- [36] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. pages 1–15, 2005.
- [37] A. Klein, C. Mannweiler, J. Schneider, and H. Schotten. Access schemes for mobile cloud computing. In *Mobile Data Management (MDM), 2010 Eleventh International Conference on*, pages 387–392, May 2010.
- [38] A. Klein, C. Mannweiler, J. Schneider, and H. D. Schotten. Access schemes for mobile cloud computing. *Mobile Data Management, IEEE International Conference on*, 0:387–392, 2010.
- [39] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In A. G. Greenberg and K. Sohrawy, editors, *INFOCOM*, pages 945–953. IEEE, 2012.
- [40] R. KT. *KVM: Paravirt-spinlock support for KVM guests*. <http://lwn.net/Articles/469918>, 2001.
- [41] KVM. <http://www.linux-kvm.org/page/Main-Page>.
- [42] H. A. Lagar-Cavilla, N. Tolia, E. de Lara, M. Satyanarayanan, and D. O’Hallaron. Interactive resource-intensive applications made easy. In *Middleware ’07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 143–163, New York, NY, USA, 2007. Springer-Verlag New York, Inc.
- [43] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Eurosys*, 2009.

- [44] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Impromptu clusters for near-interactive cloud-based services. Technical Report CSRG-TR578, Department of Computer Science, University of Toronto, 2008.
- [45] A. M. Lai, J. Nieh, B. Bohra, V. Nandikonda, A. P. Surana, and S. Varshneya. Improving web browsing performance on wireless pdas using thin-client computing. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 143–154, New York, NY, USA, 2004. ACM.
- [46] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *WIOV '11: 3rd Workshop on I/O Virtualization*, Portland, Oregon, June 2011.
- [47] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 169–180, New York, NY, USA, 2011. ACM.
- [48] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of i/o for gang scheduled workloads, 1997.
- [49] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling algorithms. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 226–236, New York, NY, USA, 1990.

- [50] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science, ExpCS '07*, New York, NY, USA, 2007.
- [51] F. Li and J. Nieh. Optimal linear interpolation coding for server-based computing. In *ICC*, 2002.
- [52] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES '01*, pages 238–246, New York, NY, USA, 2001. ACM.
- [53] Y. Ling, T. Mullen, and X. Lin. Analysis of optimal thread pool size. *SIGOPS Operating System Review*, 2000.
- [54] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, ATEC '96*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.
- [55] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, VEE '05*, pages 13–23, New York, NY, USA, 2005. ACM.
- [56] K. Z. Meth and J. Satran. Design of the iscsi protocol. In *MSS*, 2003.
- [57] A. B. Nagarajan and F. Mueller. Proactive fault tolerance for hpc with xen virtualization. In *In Proceedings of the 21st Annual International Conference*

- on *Supercomputing (ICS'07)*, pages 23–32. ACM Press, 2007.
- [58] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference*, 2005.
- [59] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *In Proc. OSDI*, pages 1–14, 2006.
- [60] S. Oaks and H. Wong. *Java Threads*. O'Reilly Media, Inc., 2004.
- [61] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *Proceedings of the First Workshop on Virtualization in Mobile Computing, MobiVirt '08*, pages 31–35, New York, NY, USA, 2008. ACM.
- [62] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10, New York, NY, USA, 2008. ACM.
- [63] OpenSSL. <https://www.openssl.org/docs/apps/speed.html>.
- [64] Oracle VM Templates. <http://www.oracle.com/technology/products/vm/templates/index.html>.
- [65] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *In Proc. of the 3rd International Conference on Distributed Computing Systems*, Ft. Lauderdale, FL, USA, October 1982.
- [66] I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt. Evaluating and optimizing thread pool strategies for real-time corba. In *LCTES*, 2001.

- [67] A. Ranadive, M. Kesavan, A. Gavrilovska, and K. Schwan. Performance implications of virtualizing multicore cluster machines. In *Proceedings of the 2Nd Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '08, pages 1–8, New York, NY, USA, 2008. ACM.
- [68] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. Optimizing virtual machine scheduling in numa multicore systems. In *Proceedings of 19th IEEE International Symposium on High Performance Computer Architecture*, HPCA '13. IEEE, 2013.
- [69] J. Rao and X. Zhou. Towards fair and efficient smp virtual machine scheduling. In *Proceedings of 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '14. ACM, 2014.
- [70] S. Research and D. Group. <http://supercluster.org/maui>.
- [71] RightScale VM Templates. <http://blog.rightscale.com/2010/03/22/rightscale-servertemplates-explained/>.
- [72] A. Roytman, A. Kansal, S. Govindan, J. Liu, and S. Nath. Pacman: Performance aware virtual machine consolidation. In *10th International Conference on Autonomic Computing (ICAC)*. USENIX, June 2013.
- [73] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving portable computer battery power through remote process execution. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(1):19–26, Jan. 1998.
- [74] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving portable computer battery power through remote process execution. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(1):19–26, Jan. 1998.

- [75] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving portable computer battery power through remote process execution. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(1):19–26, Jan. 1998.
- [76] Z. Sanaei, S. Abolfazli, A. Gani, and M. Shiraz. Sami: Service-based arbitrated multi-tier infrastructure for mobile cloud computing. *CoRR*, abs/1206.6219, 2012.
- [77] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Operating System Review*, 2002.
- [78] M. Satyanarayanan. Mobile computing: The next decade. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, pages 5:1–5:6, New York, NY, USA, 2010. ACM.
- [79] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8:14–23, 2009.
- [80] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, A. Surie, D. R. O'Hallaron, A. Wolbach, J. Harkes, A. Perrig, D. J. Farber, M. A. Kozuch, C. J. Helfrich, P. Nath, and H. A. Lagar-Cavilla. Pervasive personal computing in an internet suspend/resume system. In *IEEE Internet Computing*, 2007.
- [81] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for smp vms? In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 257–272, New York, NY, USA, 2011.

- [82] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell. Modeling virtual machine performance: Challenges and approaches. *SIGMETRICS Perform. Eval. Rev.*, 37(3):55–60, Jan. 2010.
- [83] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.
- [84] S. Vaddagiri, B. B Rao, V. Srinivasan, B. P Janakiraman Singh, and V. K Sukthankar. Scaling software on multi-core through co-scheduling of related tasks. In *in Ottawa Linux Symposium*, 2009.
- [85] VDI. <http://www.vmware.com/pdf/virtual-desktop-infrastructure-wp.pdf>.
- [86] VMware. http://www.vmware.com/pdf/vc_2_templates_usage_best_practices_wp.pdf.
- [87] VMware. *Performance best practices for VMware vSphere 4.0, VMware ESX 4.0 and ESXi 4.0*. <http://www.vmware.com/pdf/Perf-Best-Practices-vSphere4.0.pdf>, 2010.
- [88] VMware. *VMWare vSphere 4: The CPU Scheduler in VMWare ESX 4 White Paper*. <http://www.vmware.com/pdf/perf-vsphere-cpu-scheduler.pdf>, 2010.
- [89] VMWare. *Co-scheduling SMP VMs in VMware ESX server*, May,2008.

- [90] VMWare. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist.* "http://www.vmware.com/files/pdf/VMware-paravirtualization.pdf", May,2009.
- [91] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP*, 2005.
- [92] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. In *SC*, 2008.
- [93] K. Wang, J. Rao, and C.-Z. Xu. Rethink the virtual machine template. *SIGPLAN Not.*, 46:39–50, Mar. 2011.
- [94] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques, PACT '06*, pages 124–133, New York, NY, USA, 2006.
- [95] C. Weng, Z. Wang, M. Li, and X. Lu. The hybrid scheduling framework for virtual machine systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '09*, pages 111–120, New York, NY, USA, 2009.
- [96] Xen. <http://www.xen.org>.
- [97] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *SIGMETRICS Perform. Eval. Rev.*, 40(4):23–32, Apr. 2013.

- [98] S. Zachariadis, C. Mascolo, and W. Emmerich. satin: A component model for mobile self organisation. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1303–1321. Springer Berlin Heidelberg, 2004.
- [99] B. Zhao, B. C. Tak, and G. Cao. Reducing the delay and power consumption of web browsing on smartphones in 3g networks. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems, ICDCS '11*, pages 413–422, Washington, DC, USA, 2011. IEEE Computer Society.
- [100] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *VEE*, 2009.
- [101] S. Zhou and T. Brecht. Processor-pool-based scheduling for large-scale numa multiprocessors. In *SIGMETRICS*, 1991.

ABSTRACT**BUILDING A COMPUTING-AS-A-SERVICE MOBILE CLOUD
SYSTEM**

by

KUN WANG**December 2015**

Advisor: Dr. Chengzhong Xu
Major: Computer Engineering
Degree: Doctor of Philosophy

The last five years have witnessed the proliferation of smart mobile devices, the explosion of various mobile applications and the rapid adoption of cloud computing in business, governmental and educational IT deployment. There is also a growing trends of combining mobile computing and cloud computing as a new popular computing paradigm nowadays. This thesis envisions the future of mobile computing which is primarily affected by following three trends: First, servers in cloud equipped with high speed multi-core technology have been the main stream today. Meanwhile, ARM processor powered servers is growingly became popular recently and the virtualization on ARM systems is also gaining wide ranges of attentions recently. Second, high-speed internet has been pervasive and highly available. Mobile devices are able to connect to cloud anytime and anywhere. Third, cloud computing is reshaping the way of using computing resources. The classic pay/scale-as-you-go model allows hardware resources to be optimally allocated and well-managed. These three trends lend credence to a new mobile computing model with the combination of resource-rich cloud and less powerful mobile devices. In this model, mobile devices run the core vir-

tualization hypervisor with virtualized phone instances, allowing for pervasive access to more powerful, highly-available virtual phone clones in the cloud. The centralized cloud, powered by rich computing and memory recourses, hosts virtual phone clones and repeatedly synchronize the data changes with virtual phone instances running on mobile devices. Users can flexibly isolate different computing environments.

In this dissertation, we explored the opportunity of leveraging cloud resources for mobile computing for the purpose of energy saving, performance augmentation as well as secure computing enviroment isolation. We proposed a framework that allows mobile users to seamlessly leverage cloud to augment the computing capability of mobile devices and also makes it simpler for application developers to run their smartphone applications in the cloud without tedious application partitioning. This framework was built with virtualization on both server side and mobile devices. It has three building blocks including agile virtual machine deployment, efficient virtual resource management, and seamless mobile augmentation. We presented the design, implementation and evaluation of these three components and demonstrated the feasibility of the proposed mobile cloud model.

AUTOBIOGRAPHICAL STATEMENT**KUN WANG**

Kun Wang is a graduate student in the Department of Electrical and Computer Engineering at Wayne State University. He is a member of the Cloud and Internet Computing group, led by Dr. Chengzhong Xu. Prior to joining Wayne State University, he received his B.S. degrees from Huazhong University of Science and Technology in Hubei China. His research interests include virtualization, operating system, and mobile system. He has published several papers in conferences in these areas. He has also served as a peer reviewer for a few conferences and journals. Besides the academic research, he has done two internships with VMWare VMKernel team. He was one of the three best intern poster competition winners during the internship. He also served as a lab instructor for the undergraduate course Introducing Microcomputers for four consecutive semesters. He was nominated as the Outstanding Teaching Assistant of the Year in 2013.